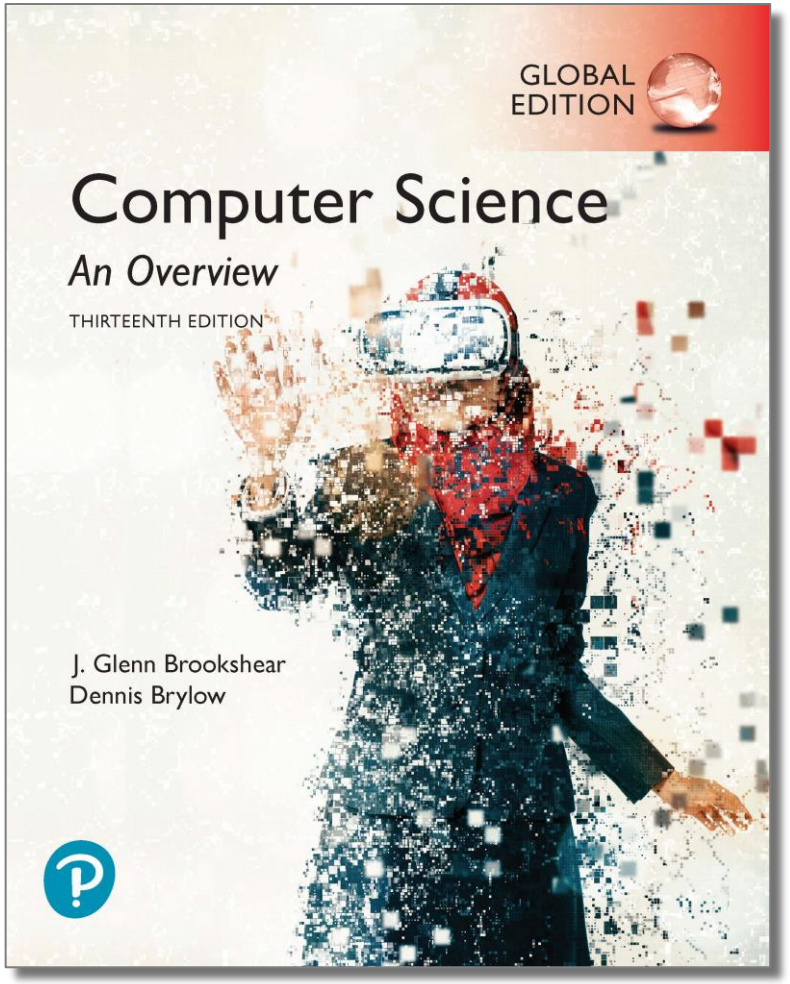
Computer Science An Overview

13th Edition, Global Edition



Chapter 6

Programming Languages

Chapter 6: Programming Languages

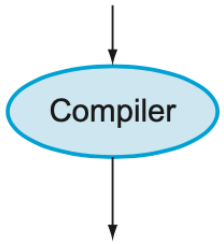
- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

6.1 Historical Perspective

- Early Generations
  - Machine Language (e.g. Vole)
  - Assembly Language
- Machine Independent Language
- Beyond – more powerful abstractions

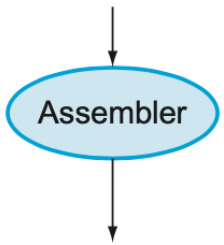
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



Binary machine language program (for MIPS)

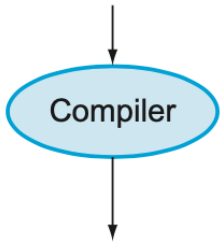
```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

6.1 Historical Perspective

- Early Generations
  - Machine Language (e.g. Vole)
  - Assembly Language
- Machine Independent Language
- Beyond – more powerful abstractions

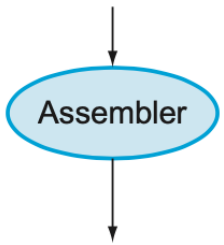
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



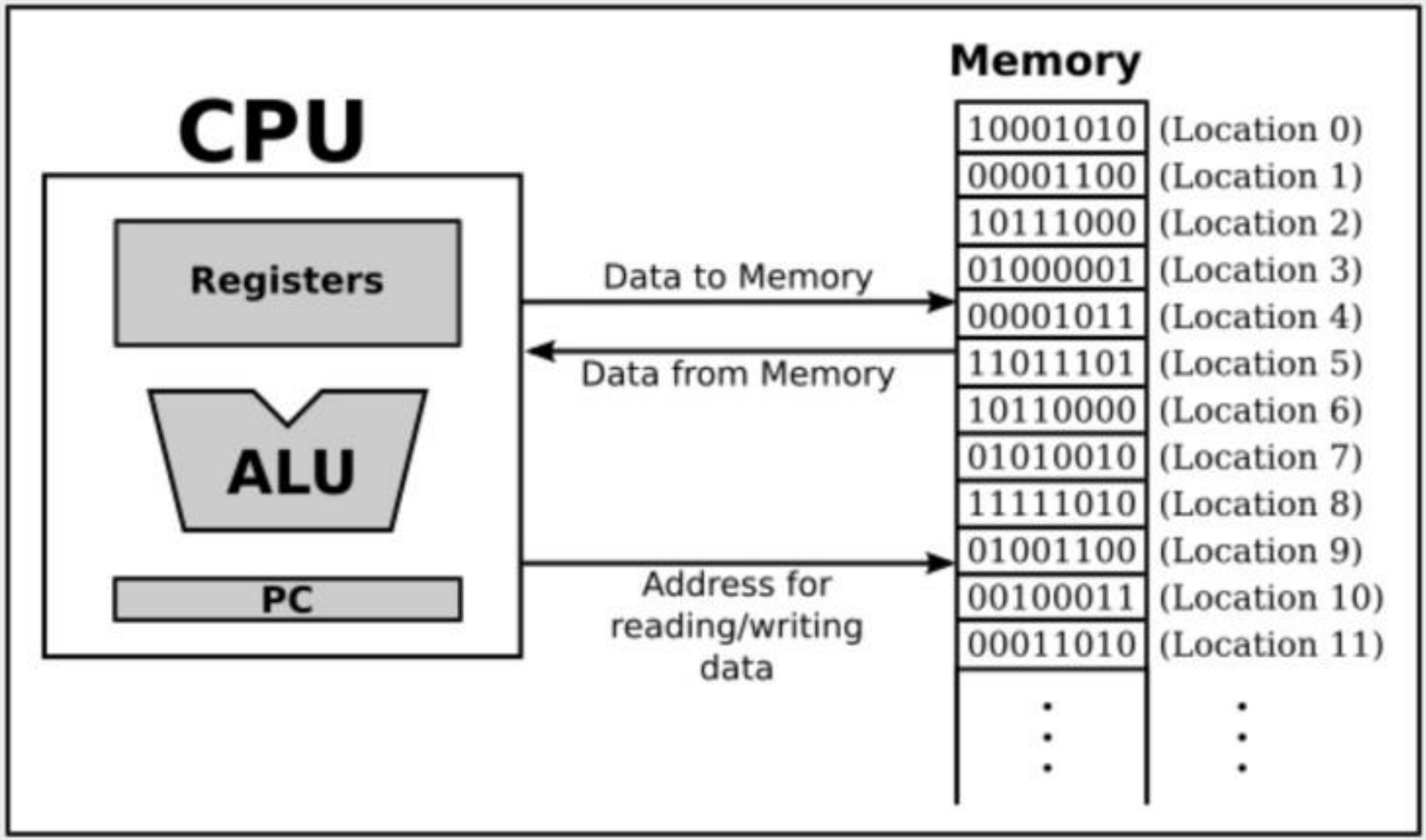
Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



Binary machine language program (for MIPS)

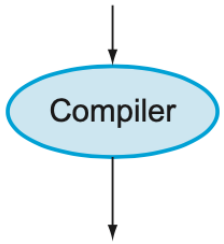
```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```



https://sciencesoftcode.wordpress.com/2020/08/14/programming-1-java/

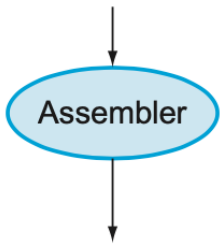
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



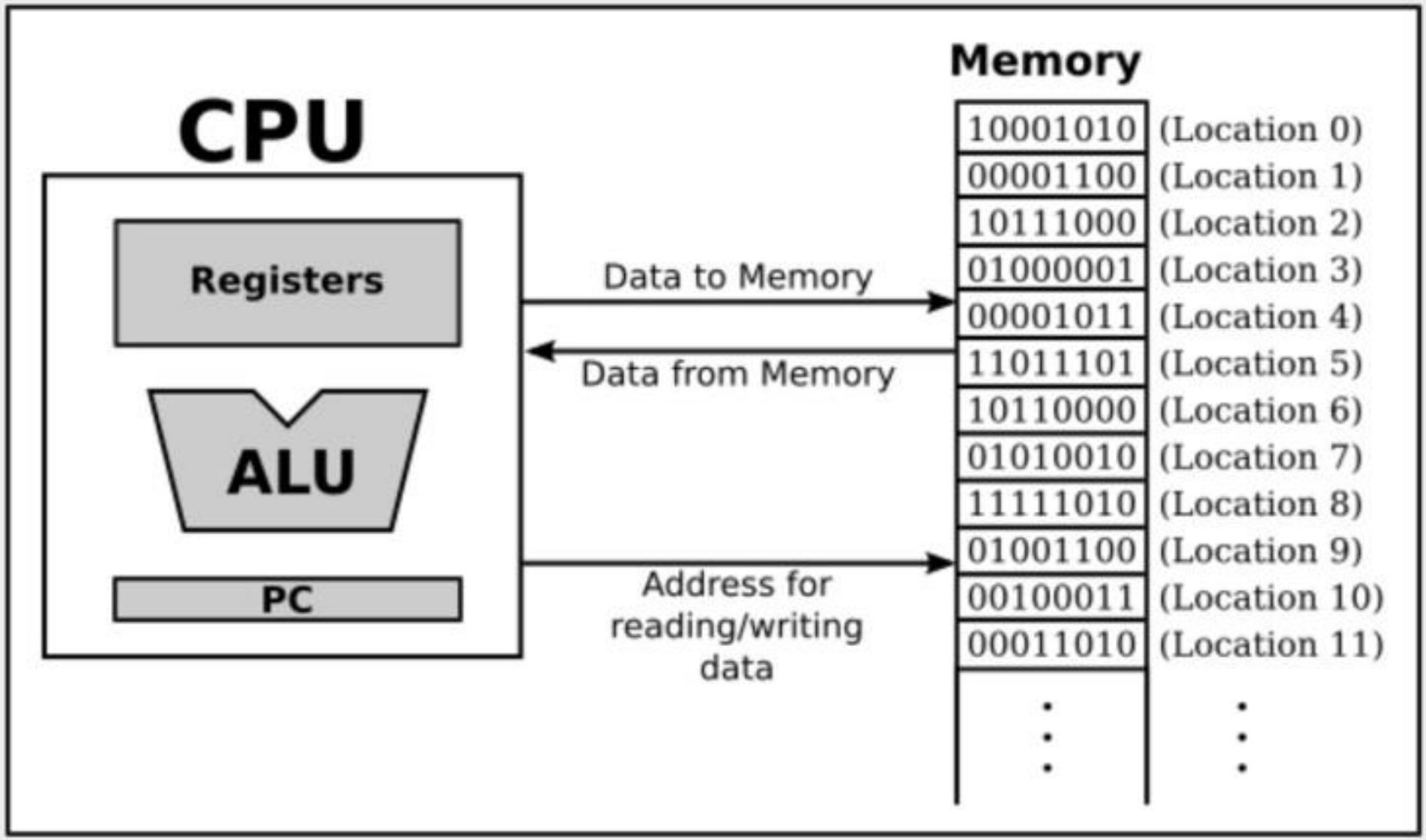
Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```



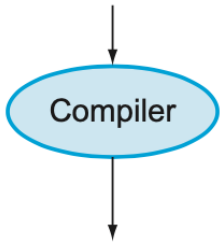
https://sciencesoftcode.wordpress.com/2020/08/14/programming-1-java/

6.1 Historical Perspective

- Early Generations
  - Machine Language (e.g. Vole)
  - Assembly Language
- Machine Independent Language
- Beyond – more powerful abstractions

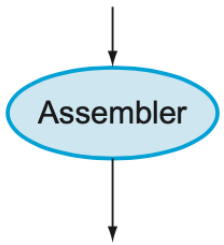
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



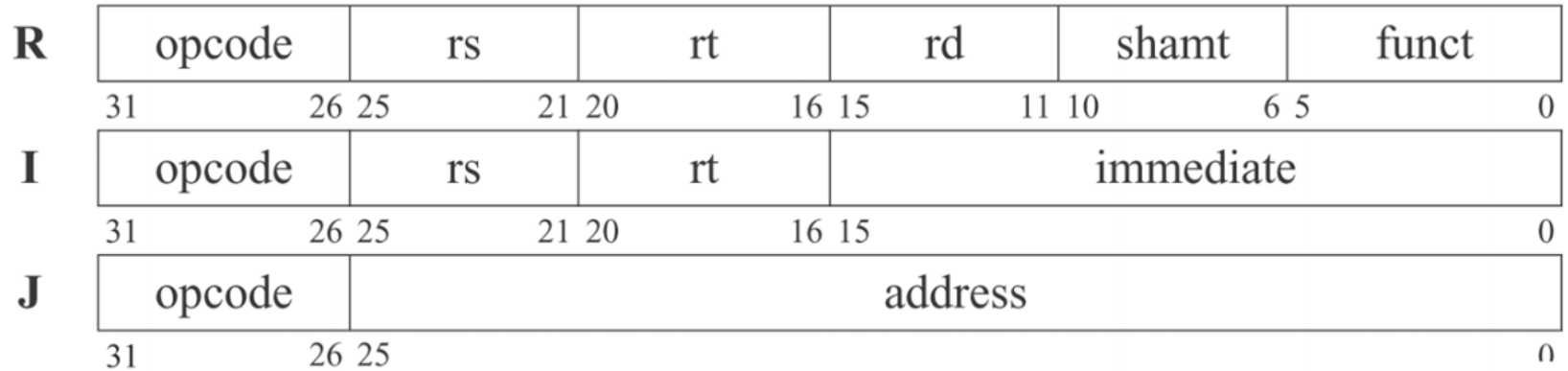
Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Second-generation: Assembly language

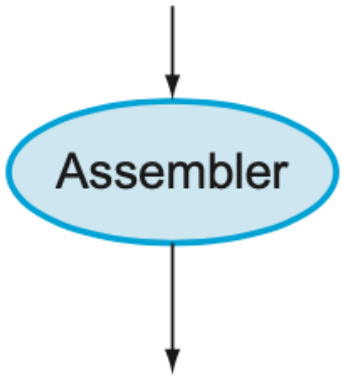
- A mnemonic system for representing machine instructions
  - Mnemonic names for op-codes
  - Program **variables** or **identifiers**: Descriptive names for memory locations, chosen by the programmer

BASIC INSTRUCTION FORMATS



Assembly language program (for MIPS)

```
swap:
        multi  $2, $5,4
        add    $2, $4,$2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```



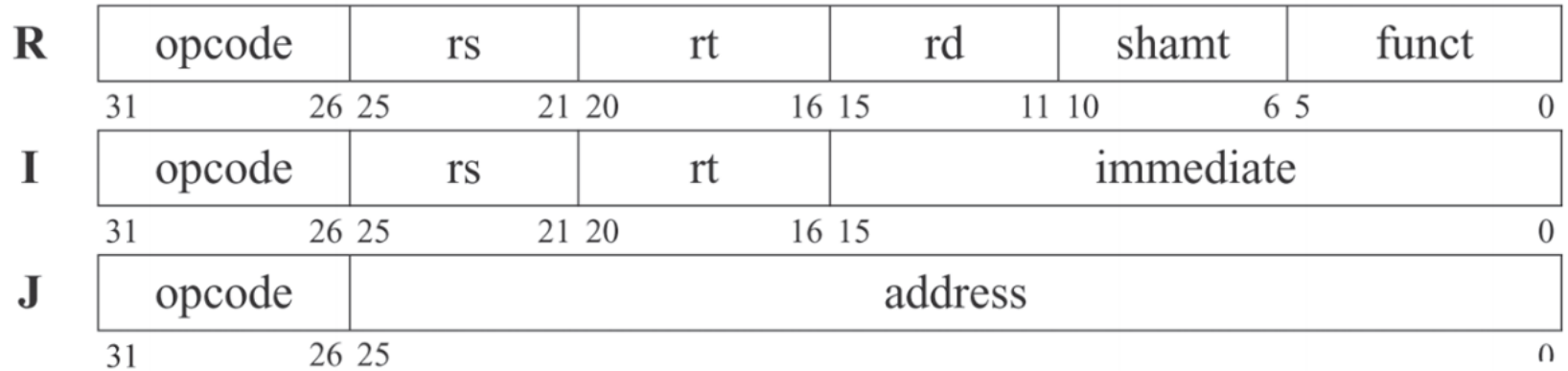
Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Second-generation: Assembly language

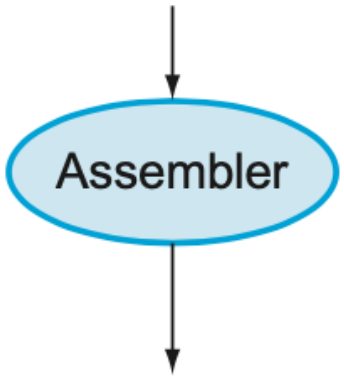
- A mnemonic system for representing machine instructions
  - Mnemonic names for op-codes
  - Program **variables** or **identifiers**: Descriptive names for memory locations, chosen by the programmer

BASIC INSTRUCTION FORMATS



Assembly language program (for MIPS)

```
swap:
        multi  $2, $5,4
        add    $2, $4,$2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```



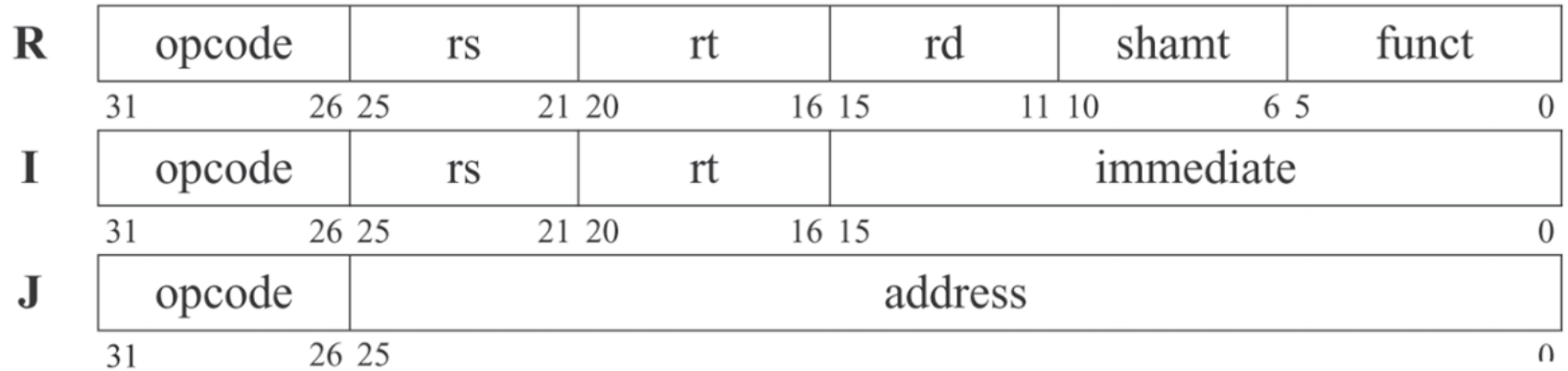
Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Second-generation: Assembly language

- A mnemonic system for representing machine instructions
  - Mnemonic names for op-codes
  - Program **variables** or **identifiers**: Descriptive names for memory locations, chosen by the programmer

BASIC INSTRUCTION FORMATS

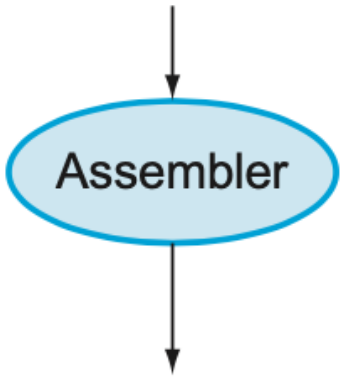


Assembly Language Characteristics

- One-to-one correspondence between machine instructions and assembly instructions
  - Programmer must think like the machine
- Inherently machine-dependent
- Converted to machine language by a program called an **assembler**

Assembly language program (for MIPS)

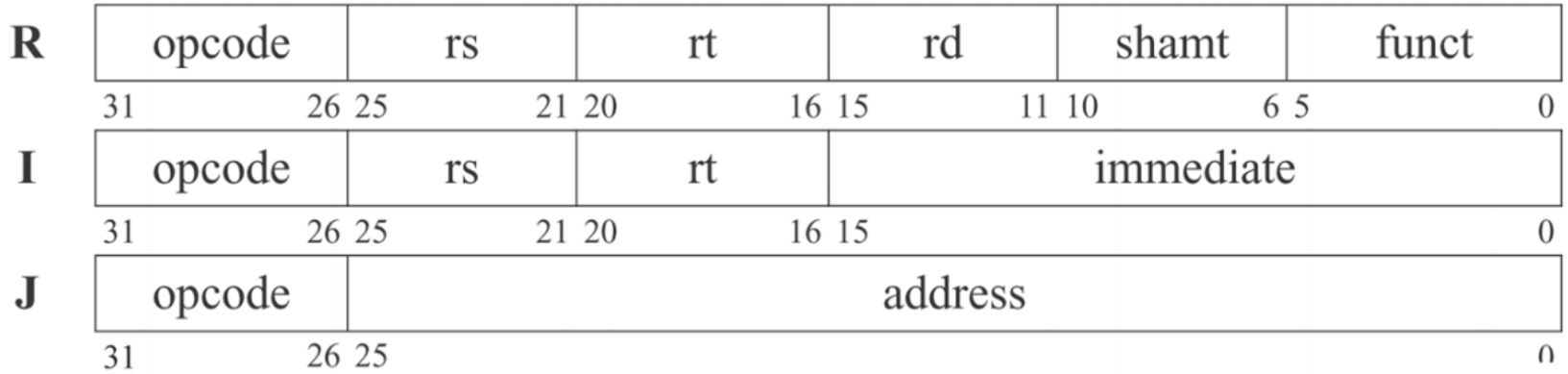
```
swap:
        multi  $2, $5,4
        add    $2, $4,$2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```



Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

BASIC INSTRUCTION FORMATS



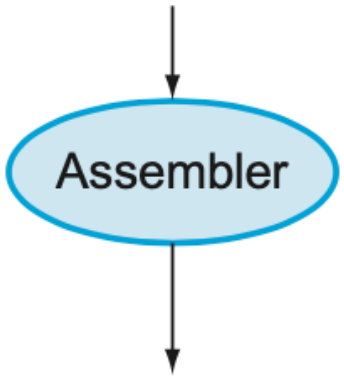
https://stackoverflow.com/questions/42392369/why-are-opcode-field-and-funct-field-apart-in-mips

Assembly Language Characteristics

- One-to-one correspondence between machine instructions and assembly instructions
  - Programmer must think like the machine
- Inherently machine-dependent
- Converted to machine language by a program called an **assembler**

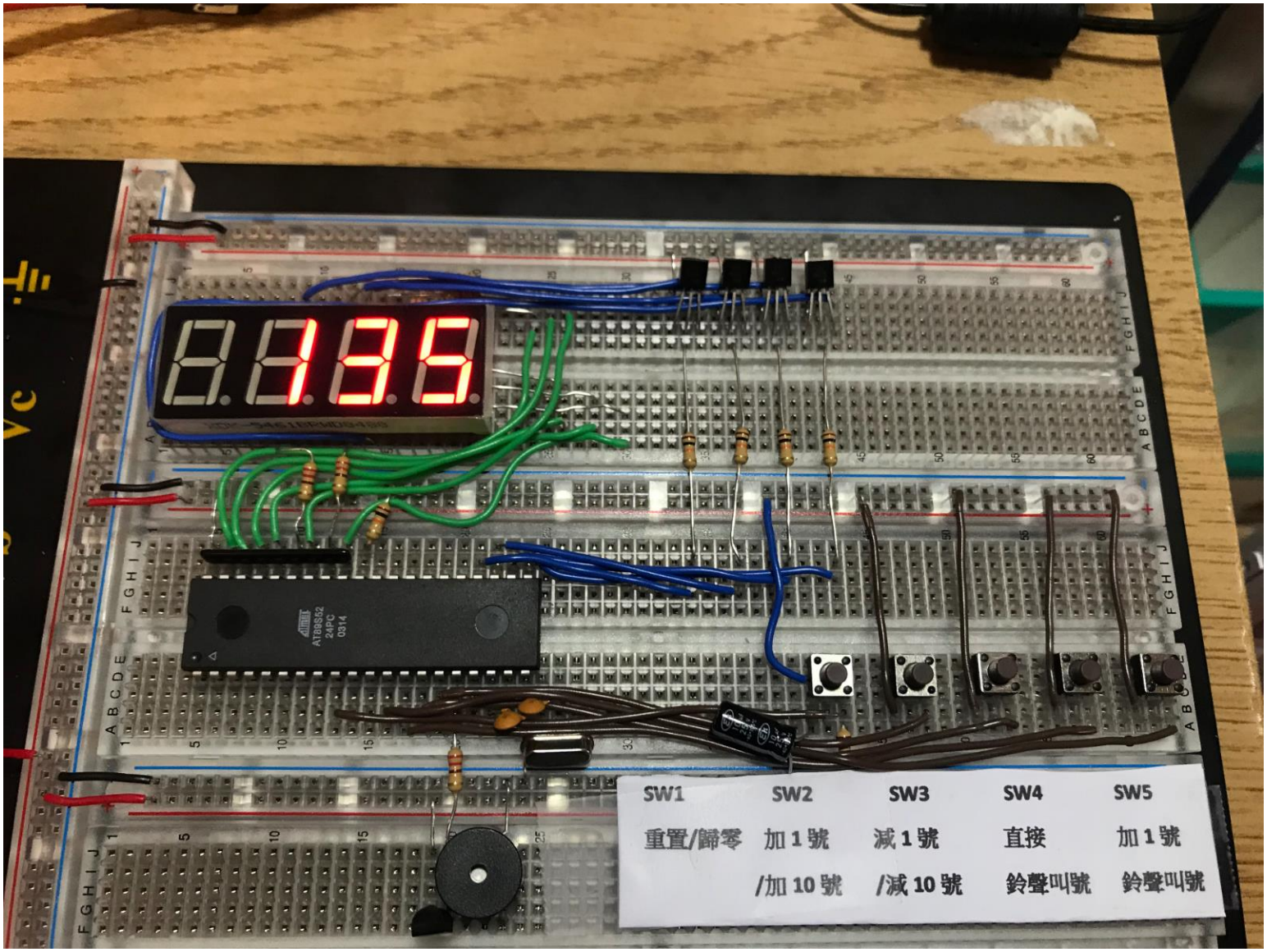
Assembly language program (for MIPS)

```
swap:
        multi  $2, $5,4
        add    $2, $4,$2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```



Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```



https://www.cakeresume.com/portfolios/8051?locale=zh-TW

Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Examples: FORTRAN, COBOL
- Each primitive corresponds to a sequence of machine language instructions
- Converted to machine language by a program called a **compiler**

Algorithm 1 Consensus-Based FA

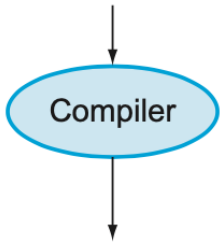
procedure CFA($\mathcal{N}_{\bar{k}}, \epsilon_t, \alpha_{t,i}$)
  initialize $\mathbf{W}_{0,k} \leftarrow$ device k
  for each round $t = 1, 2, \ldots,$ **do** ▷ Main loop
    receive $\{\mathbf{W}_{t,i}\}_{i \in \mathcal{N}_{\bar{k}}}$ ▷ RX from neighbors
    $\boldsymbol{\psi}_{t,k} \leftarrow \mathbf{W}_{t,k}$
    for all devices $i \in \mathcal{N}_{\bar{k}}$ **do**
      $\boldsymbol{\psi}_{t,k} \leftarrow \boldsymbol{\psi}_{t,k} + \epsilon_t \alpha_{t,i} (\mathbf{W}_{t,i} - \mathbf{W}_{t,k})$
    end for
    $\mathbf{W}_{t+1,k} = \text{ModelUpdate}(\boldsymbol{\psi}_{t,k})$
    send$(\mathbf{W}_{t+1,k})$ ▷ TX to neighbors
  end for
end procedure
procedure MODELUPDATE($\boldsymbol{\psi}_{t,k}$) ▷ Local SGD
  $\mathcal{B} \leftarrow$ mini-batches of size B
  for batch $b \in \mathcal{B}$ **do** ▷ Local model update
    $\boldsymbol{\psi}_{t,k} \leftarrow \boldsymbol{\psi}_{t,k} - \mu_t \nabla L_{t,k}(\boldsymbol{\psi}_{t,k})$
  end for
  $\mathbf{W}_{t,k} \leftarrow \boldsymbol{\psi}_{t,k}$
  return$(\mathbf{W}_{t,k})$
end procedure

Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Examples: FORTRAN, COBOL
- Each primitive corresponds to a sequence of machine language instructions
- Converted to machine language by a program called a **compiler**

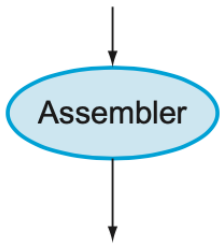
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Figure 6.1 Generations of programming languages

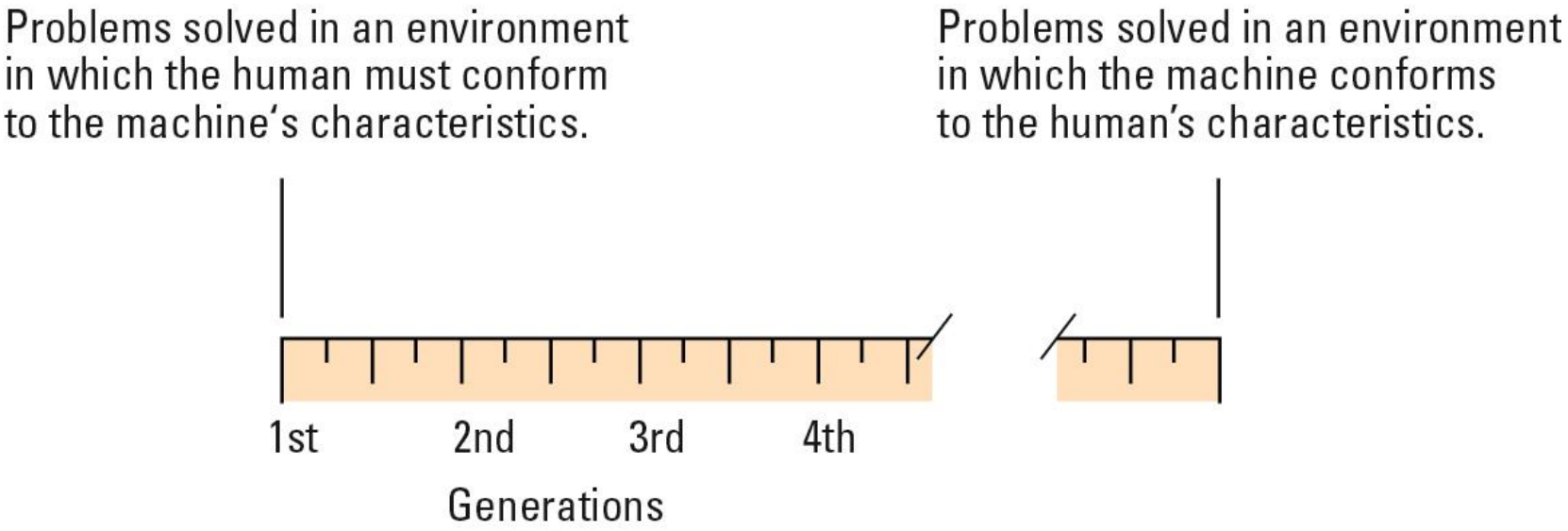
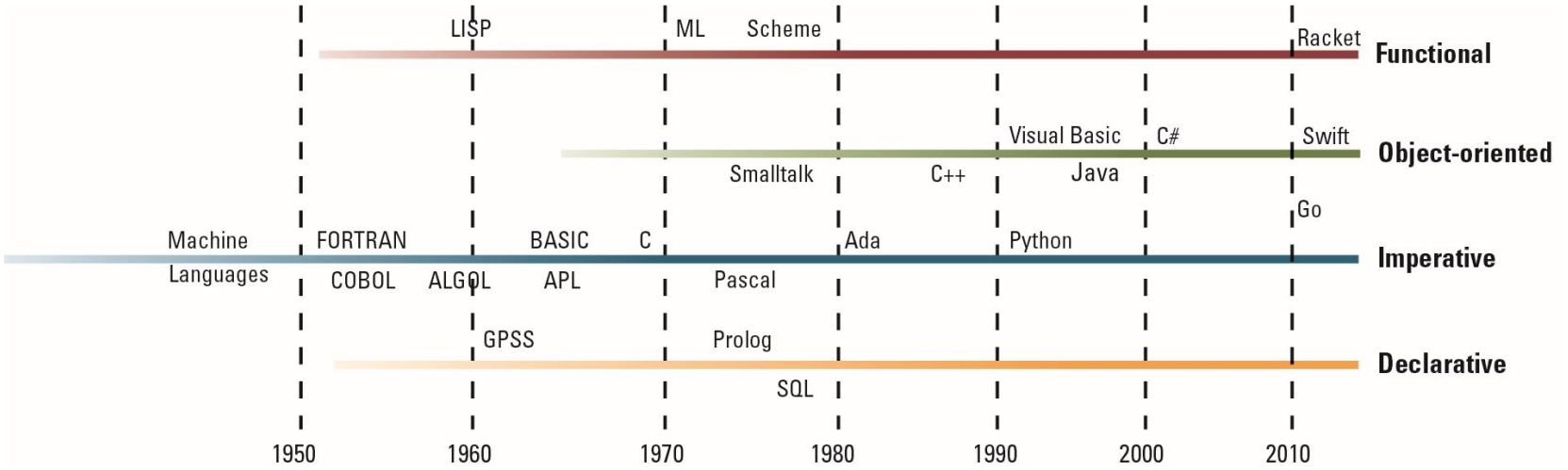


Figure 6.2 The evolution of programming paradigms



例子 [編輯]

彈簧 [編輯]

把質量為*M*的物體懸掛在勁度係數為*k*的彈簧的底端，則物體將進行簡諧運動，其方程式為：

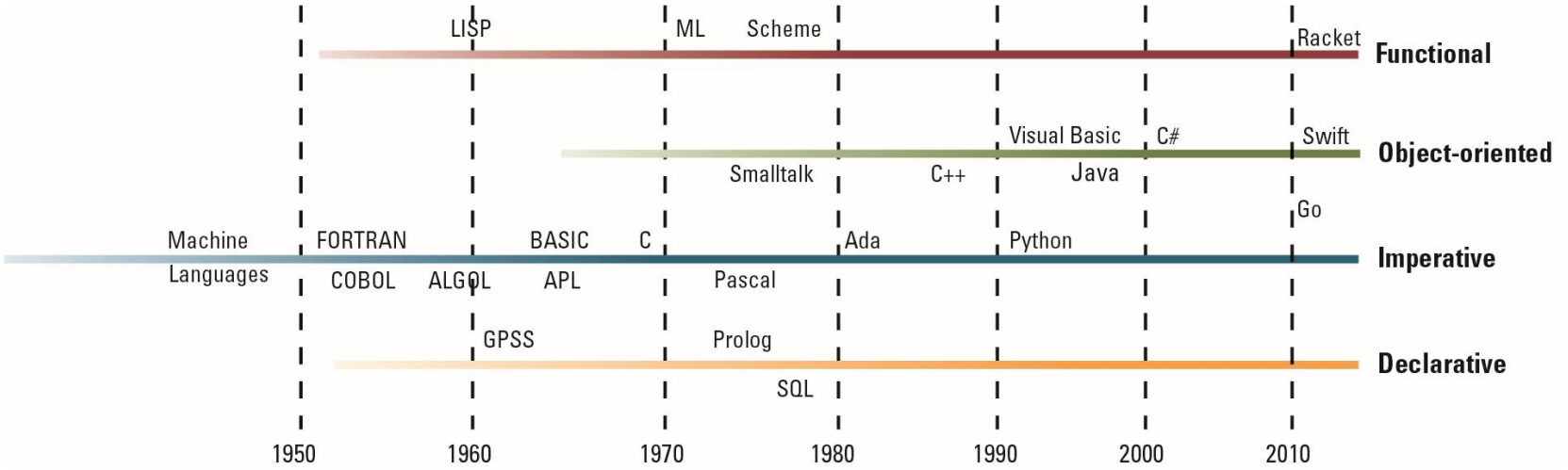
$$\omega = 2\pi f = \sqrt{\frac{k}{M}}.$$

如果要計算它的週期，可以用以下的公式：

$$T = \frac{1}{f} = 2\pi\sqrt{\frac{M}{k}}\ .$$

https://zh.wikipedia.org/zh-tw/%E7%B0%A1%E8%AB%A7%E9%81%8B%E5%8B%95

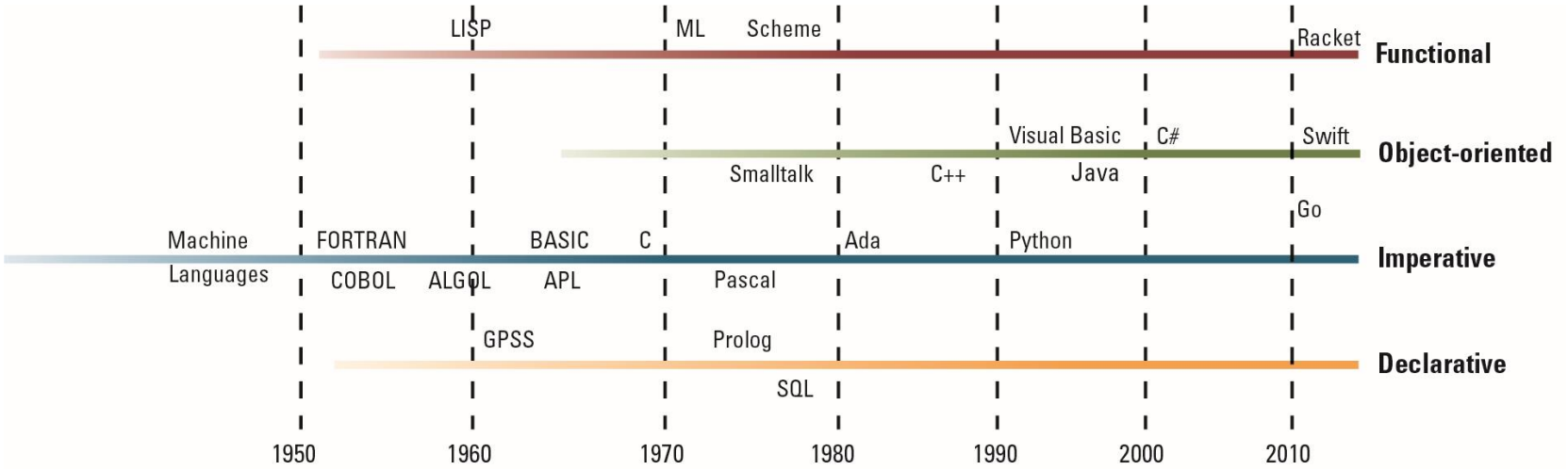
Figure 6.2 The evolution of programming paradigms



Chapter 6: Programming Languages

- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

Figure 6.2 The evolution of programming paradigms



https://www.youtube.com/watch?v=pSoDUFdqVOU&t=479s
https://www.youtube.com/watch?v=3EkaxkNGXD8

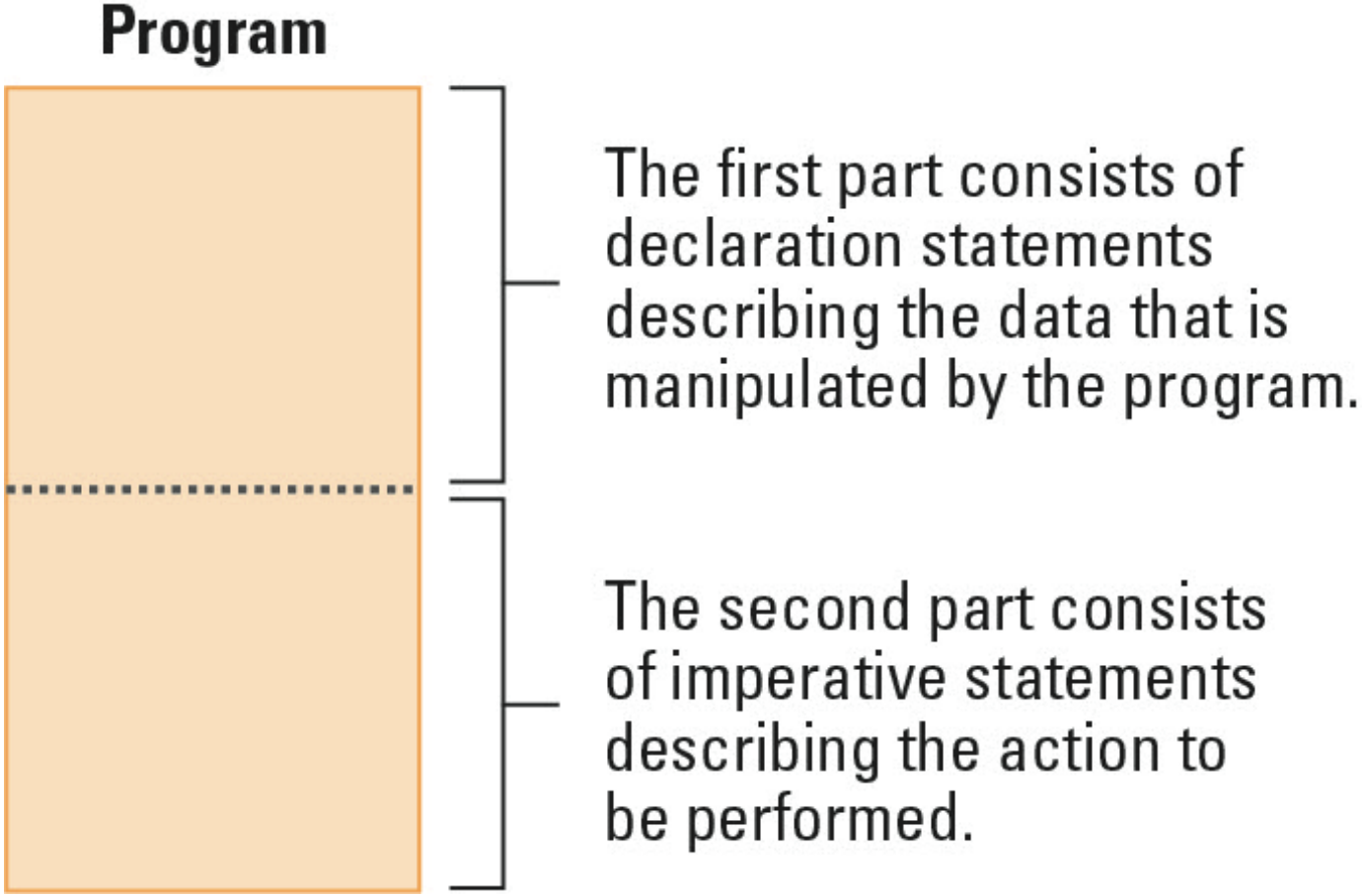
6.2 Traditional Programming Concepts

- Variables and Data Types
- Data Structure
- Constants and Literals
- Assignment Statements
- Control Statements
- Comments

6.2 Traditional Programming Concepts

- High-level languages (C, C++, Java, C#, FORTRAN) include many kinds of abstractions
  - Simple: constants, literals, variables
  - Complex: statements, expressions, control
  - Esoteric: procedures, modules, libraries

Figure 6.4 The composition of a typical imperative program or program unit



Process (其他版本 = jobs = task (Linux))

(一) Def:

A program in execution, 即執行中的程式

當 Process 被建立後, Process 的主要組成包含下列:

1. Code section (or Text section), 即 program code
2. Data section: containing the global variables
3. Stack: 包含 temporary data
   (e.g. Function parameters, Local variables, return address)
4. Heap: 包含 memory dynamically allocated during run time)
   (e.g. pointer 變數之要求空間)
5. Programming Counter and other register
   內放下一條指令之所在位址

Process (其他版本 = jobs = task (Linux))

(一) Def:

A program in execution, 即執行中的程式

當 Process 被建立後, Process 的主要組成包含下列:

1. Code section (or Text section), 即 program code
2. Data section: containing the global variables
3. Stack: 包含 temporary data
   (e.g. Function parameters, Local variables, return address)
4. Heap: 包含 memory dynamically allocated during run time)
   (e.g. pointer 變數之要求空間)
5. Programming Counter and other register
   內放下一條指令之所在位址

(=) Process V.S. Program

Process	Program
A program in execution	Just a file stored in storage device
"Active" entity	"Passive" entity

圖示：

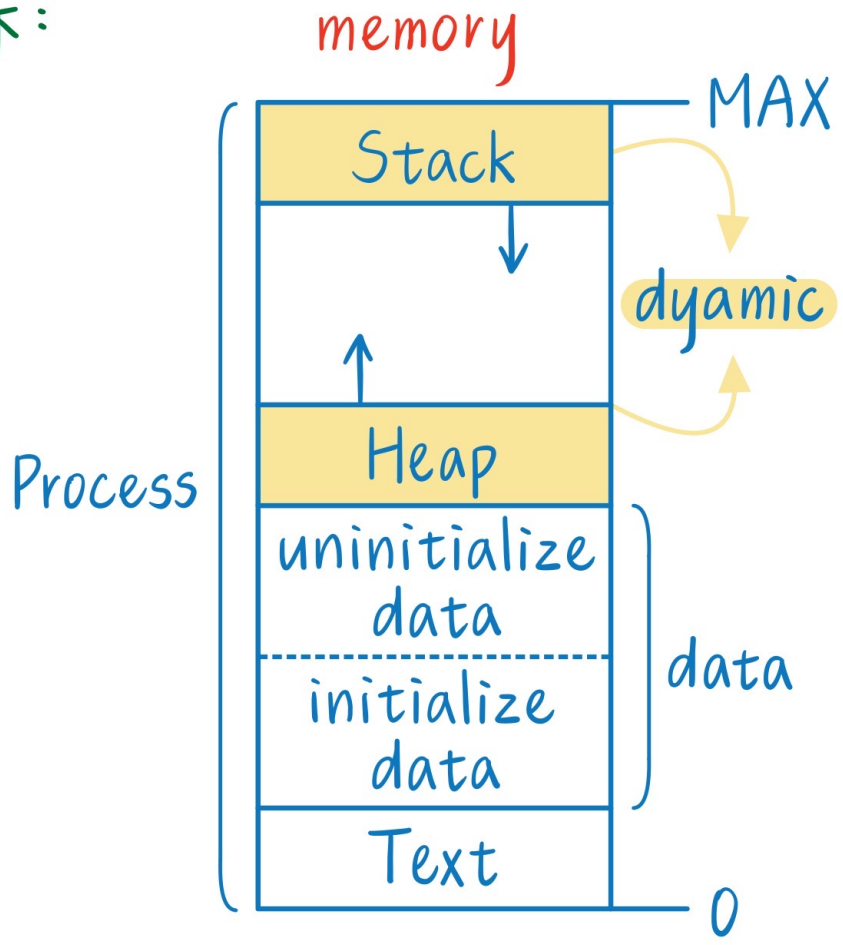
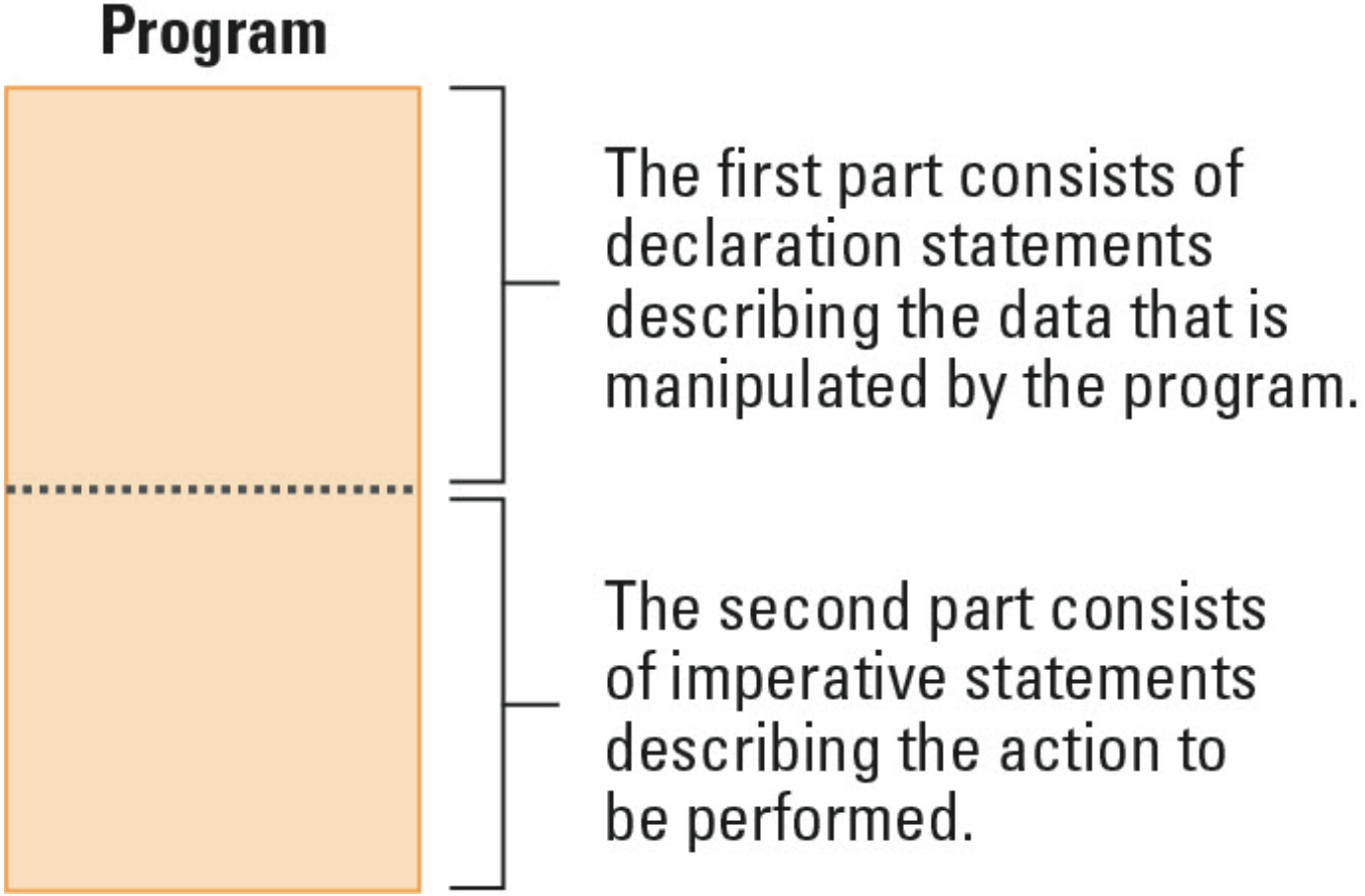

memory
MAX
Stack
dyamic
Heap
Process
uninitialize data
initialize data
data
Text
0

Figure 6.4 The composition of a typical imperative program or program unit



Data Types

- Integer: Whole numbers
- Real (float): Numbers with fractions
- Character: Symbols
- Boolean: True/false

Variables and Data types

```
float Length, Width;
int   Price, Total, Tax;
char  Symbol;

int   WeightLimit = 100;
```

6.2 Traditional Programming Concepts

- High-level languages (C, C++, Java, C#, FORTRAN) include many kinds of abstractions
  - Simple: constants, literals, variables
  - Complex: statements, expressions, control
  - Esoteric: procedures, modules, libraries

Variables and Data types

```
float Length, Width;
int   Price, Total, Tax;
char  Symbol;

int   WeightLimit = 100;
```

Data Structure

- Conceptual shape or arrangement of data
- A common data structure is the **array**
  - In C

```
int Scores[2][9];
```

  - In FORTRAN

```
INTEGER Scores(2,9)
```

Figure 6.5 A two-dimensional array with two rows and nine columns

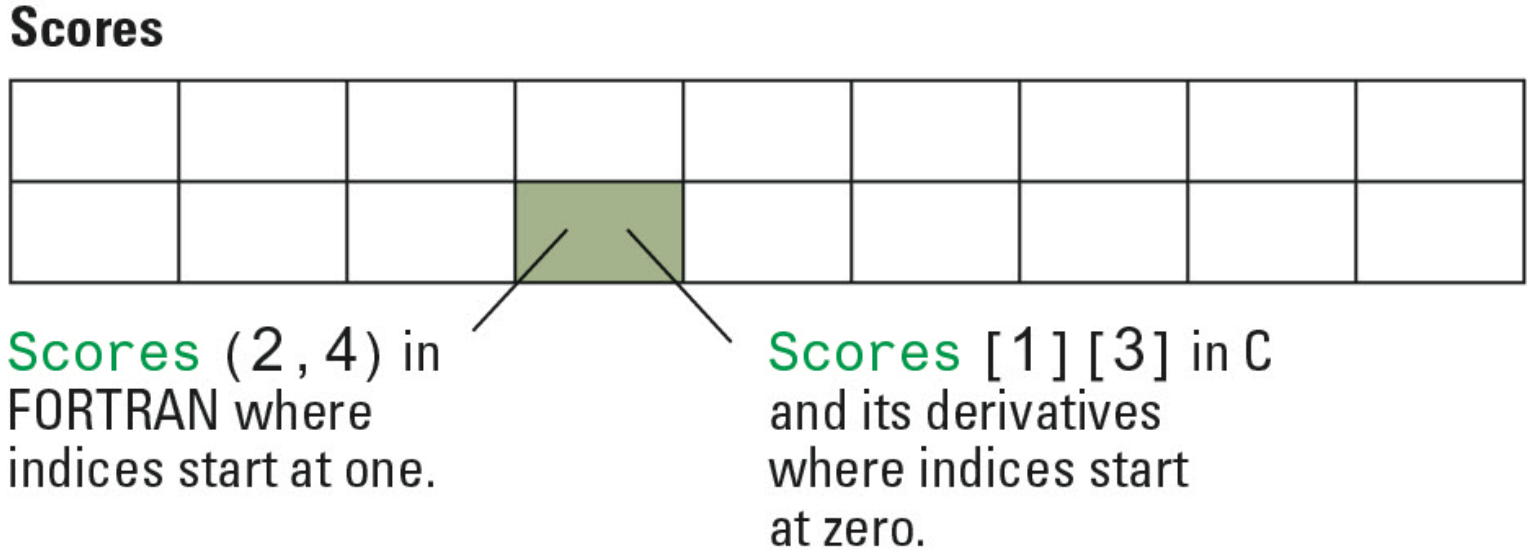
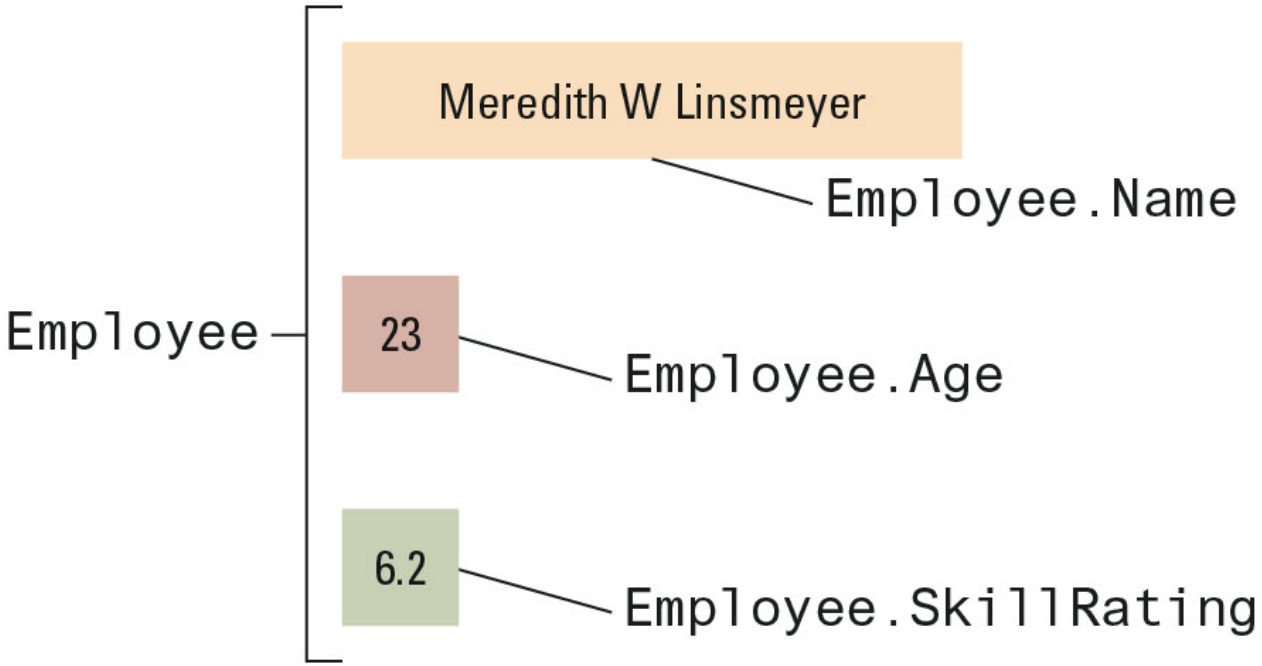


Figure 6.6 The conceptual structure of the aggregate type Employee




```
struct {    char  Name[25];
            int   Age;
            float SkillRating;
          } Employee;
```

Assignment Statements

- In C, C++, C#, Java

```
Z = X + y;
```

- In Ada

```
Z := X + y;
```

- In APL (A Programming Language)

```
Z ← X + y
```

Control Statements

- Go to statement

```
        goto 40
    20  Evade()
        goto 70
    40  if (KryptoniteLevel < LethalDose) then goto
60
        goto 20
    60  RescueDamsel()
    70  ...
```

- As a single statement

```
if (KryptoniteLevel < LethalDose):
    RescueDamsel()
else:
    Evade()
```

Control Statements (continued)

- If in Python

```
if (condition):
    statementA
else:
    statementB
```

- In C, C++, C#, and Java

```
if (condition) statementA; else statementB;
```

- In Ada

```
IF condition THEN
    statementA;
ELSE
    statementB;
END IF;
```

Control Statements (continued)

- While in Python

```
while (condition):
    body
```

- In C, C++, C#, and Java

```
while (condition)
{ body }
```

- In Ada

```
WHILE condition LOOP
    body
END LOOP;
```

Control Statements (continued)

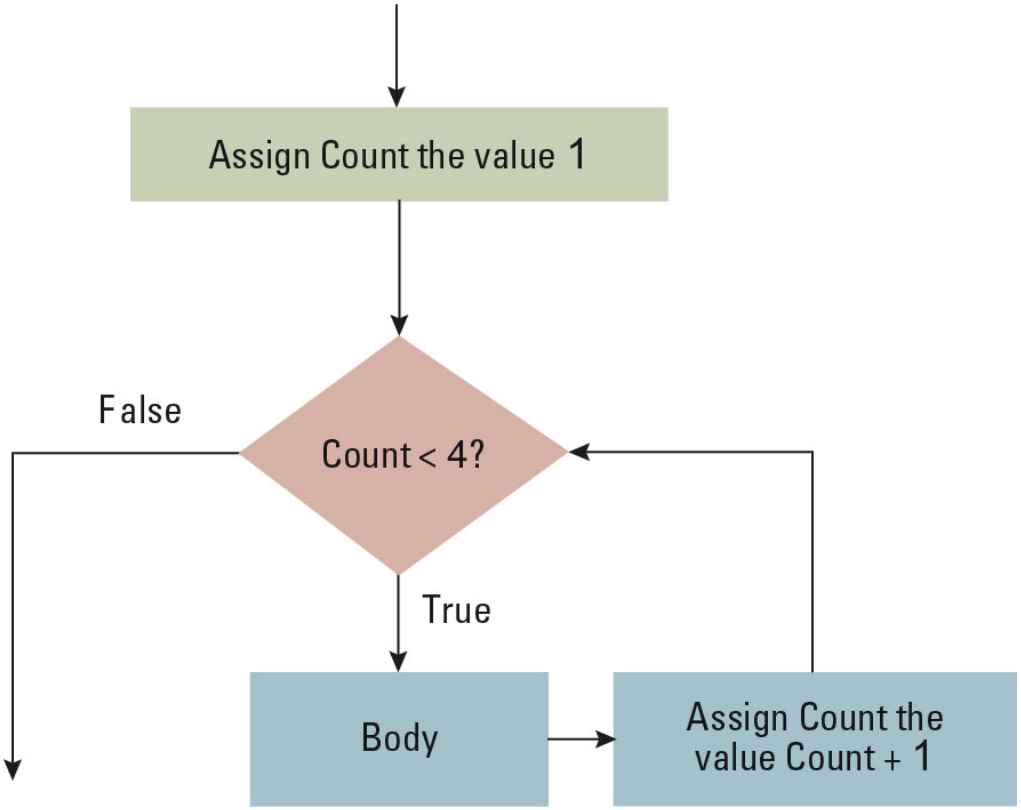
- Switch statement in C, C++, C#, and Java

```
switch (variable) {
    case 'A': statementA; break;
    case 'B': statementB; break;
    case 'C': statementC; break;
    default:  statementD; }
```

- In Ada

```
CASE variable IS
    WHEN 'A'=> statementA;
    WHEN 'B'=> statementB;
    WHEN 'C'=> statementC;
    WHEN OTHERS=> statementD;
END CASE;
```

Figure 6.7 The for loop structure and its representation in C++, C#, and Java




```
for (int Count = 1; Count < 4; Count++)
    body;
```

Comments

- Explanatory statements within a program
- Helpful when a human reads a program
- Ignored by the compiler

```
/* This is a comment in C/C++/Java. */

// This is a comment in C/C++/Java.
```

6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

```
# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = v0_A * t_A + (0.5) * a_A * (t_A^2)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_B = v0_B * t_B + (0.5) * a_B * (t_B^2)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_C = v0_C * t_C + (0.5) * a_C * (t_C^2)
```

```python
# 距離公式
def vt_func(v0, a, t):
    s = v0 * t + (0.5) * a * (t^2)
    return s

# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = vt_func(v0_A, a_A, t_A)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_A = vt_func(v0_B, a_B, t_B)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_A = vt_func(v0_C, a_C, t_C)
```

```
# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = v0_A * t_A + (0.5) * a_A * (t_A^2)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_B = v0_B * t_B + (0.5) * a_B * (t_B^2)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_C = v0_C * t_C + (0.5) * a_C * (t_C^2)
```

```python
# 距離公式
def vt_func(v0, a, t):
    s = v0 * t + (0.5) * a * (t^2)
    return s

# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = vt_func(v0_A, a_A, t_A)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_A = vt_func(v0_B, a_B, t_B)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_A = vt_func(v0_C, a_C, t_C)
```

6.3 Procedural Units

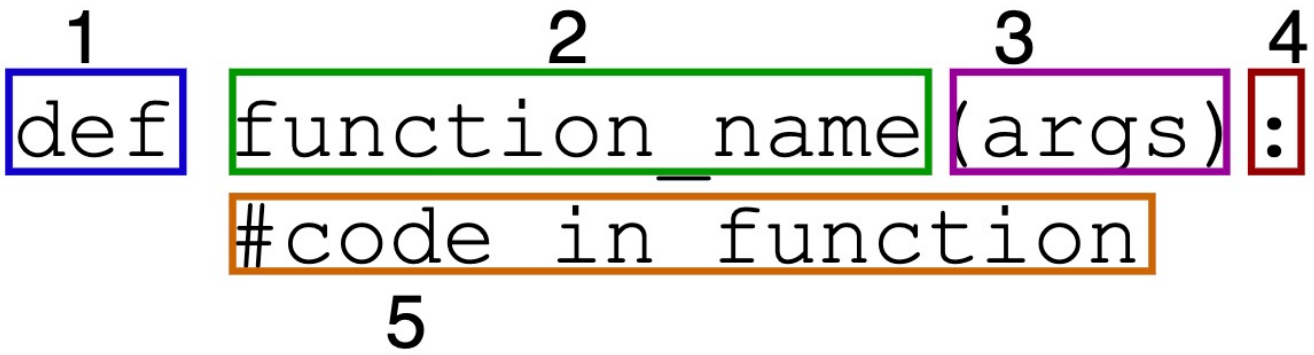
- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

Creating a Function

To create a function, you would have to first write a function header. A function header looks like this:

```
def function_name(args):
    #code in function
```

Let's break it down:



1. The `def` keyword just indicates that we are starting to write a function header
2. The name of the function goes here. You can name a function in the same ways you can name a variable.
3. Arguments go here. This is covered in the next lesson, so here you would usually leave it as blank parentheses `()`
4. Colon is required! Don't forget!
5. The code underneath that is tabbed is the code inside the function.

https://teachen.info/cspp/unit4/u0401-functions.html

6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

```python
# 距離公式
def vt_func(v0, a, t):
    s = v0 * t + (0.5) * a * (t^2)
    return s

# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = vt_func(v0_A, a_A, t_A)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_A = vt_func(v0_B, a_B, t_B)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_A = vt_func(v0_C, a_C, t_C)
```

6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

Figure 6.8 The flow of control involving a function

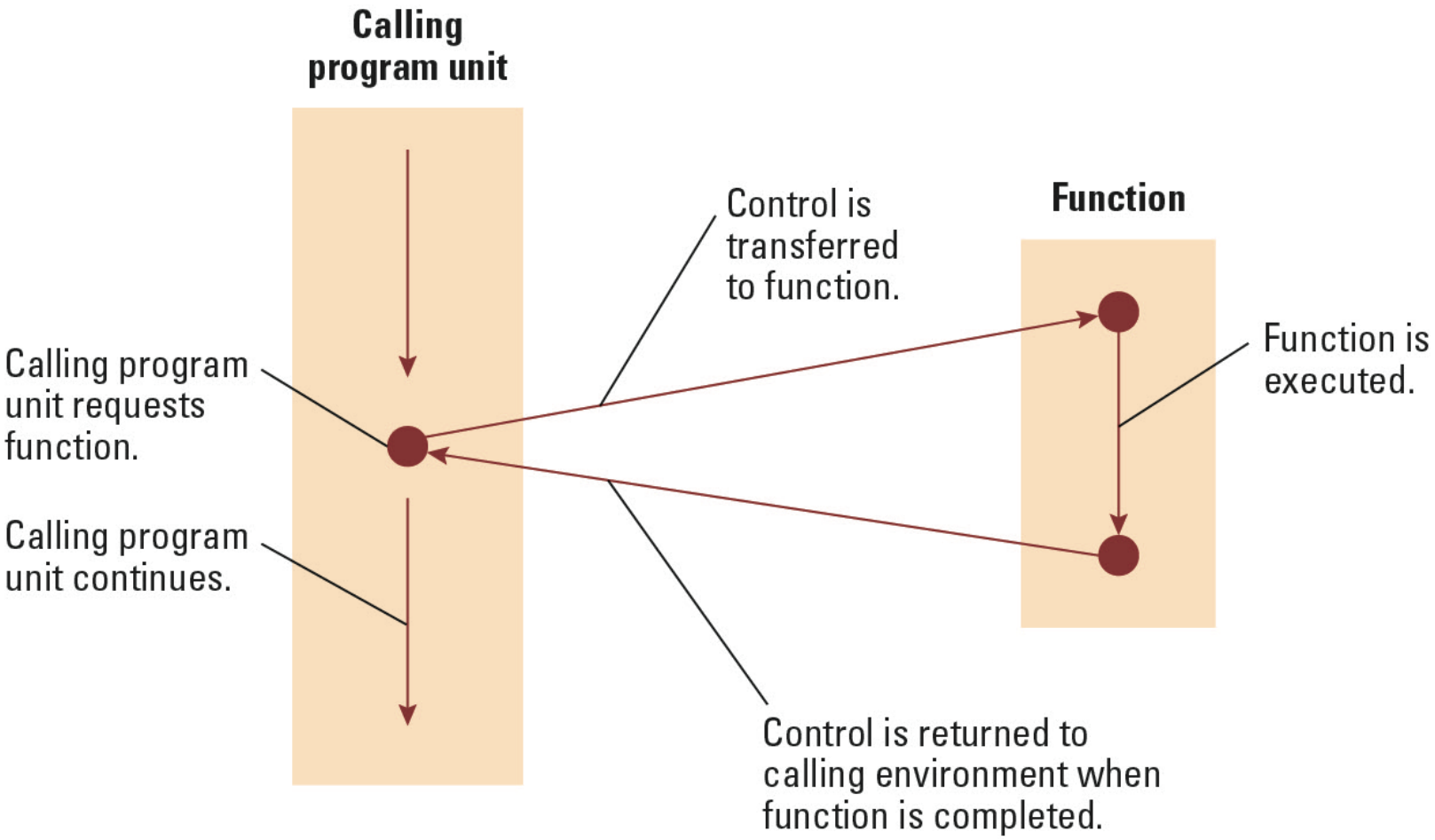


Figure 6.9 The function ProjectPopulation written in the programming language C

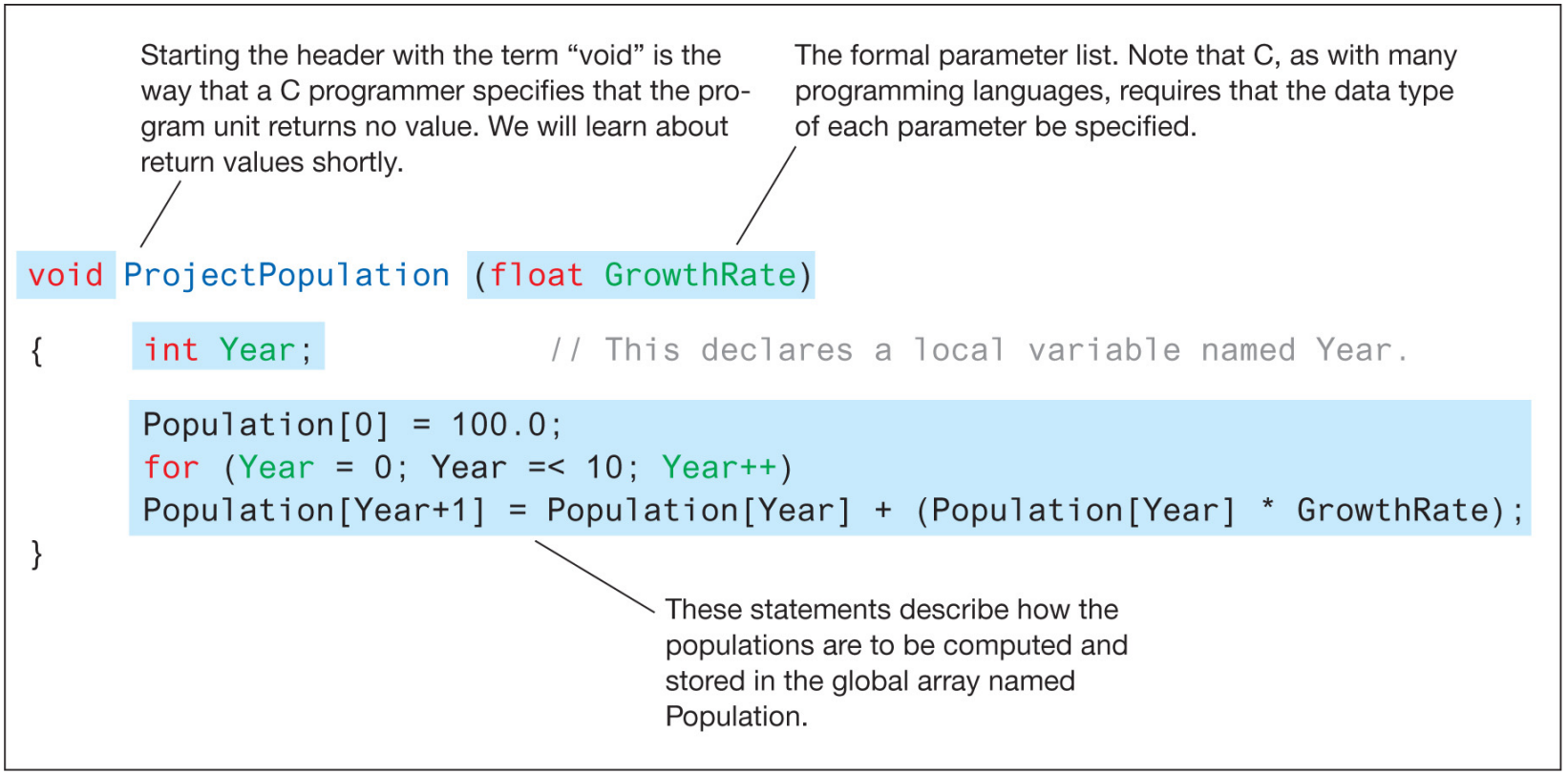


Figure 6.12 The fruitful function CylinderVolume written in the programming language C

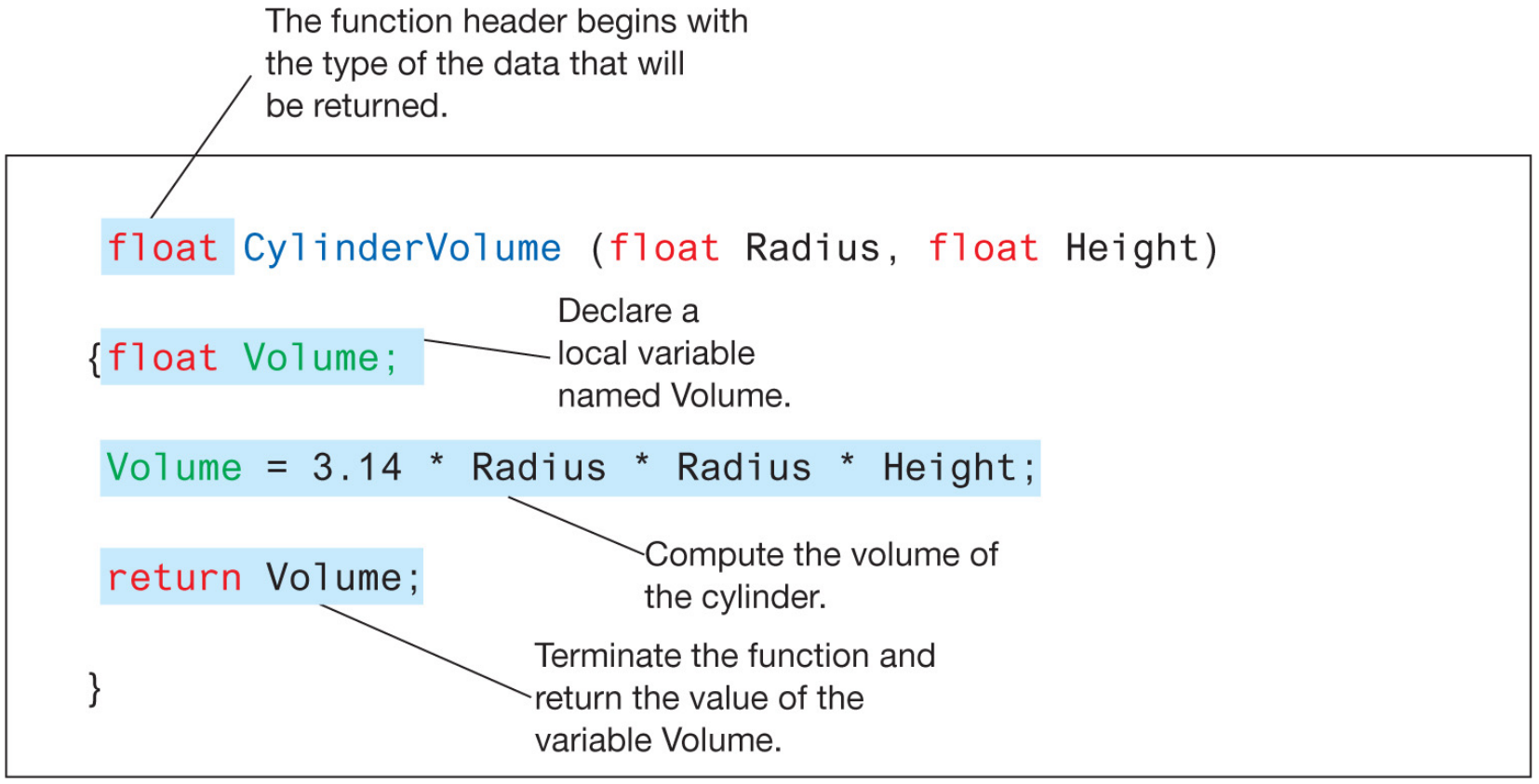


Figure 6.9 The function ProjectPopulation written in the programming language C

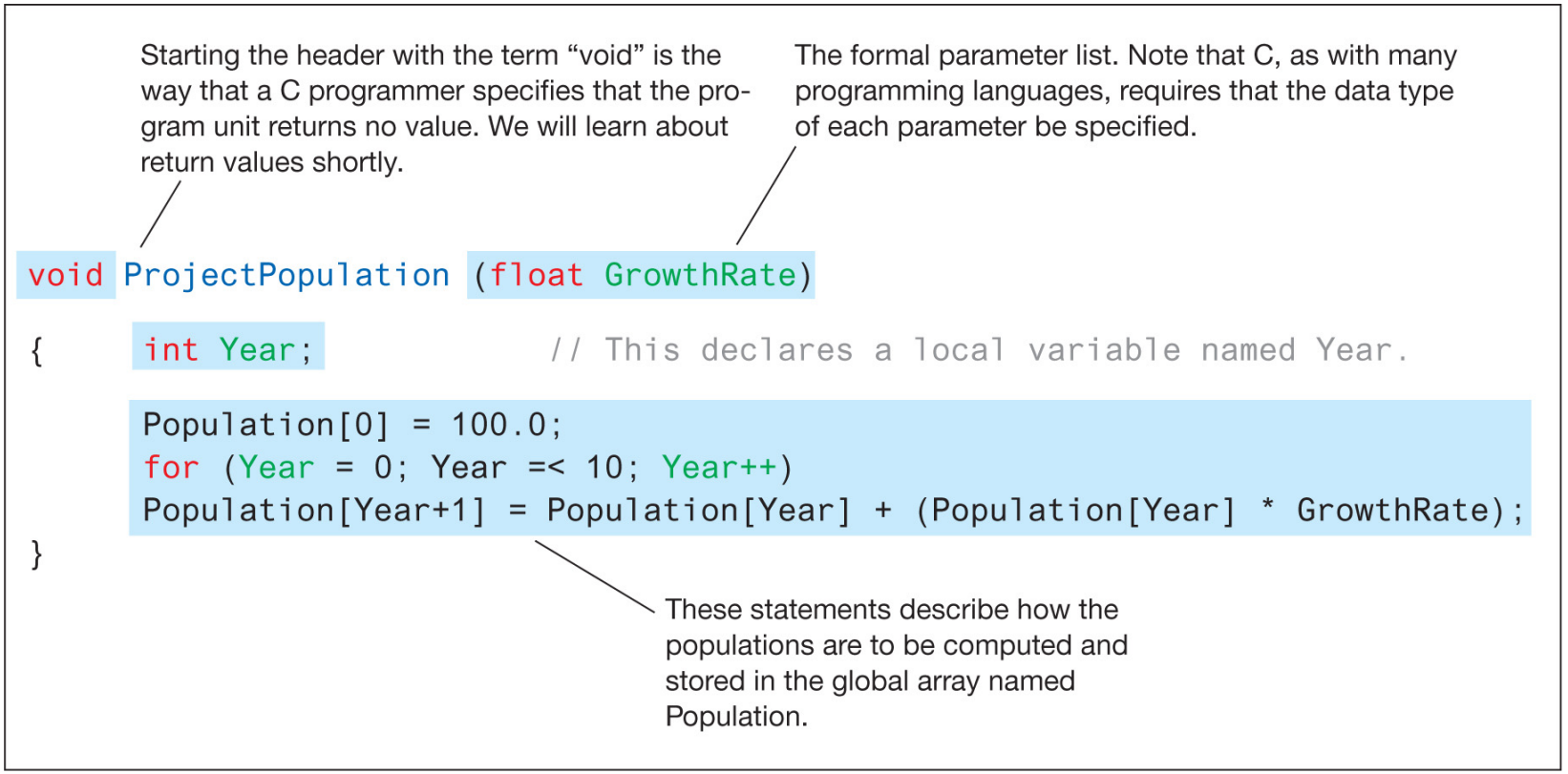
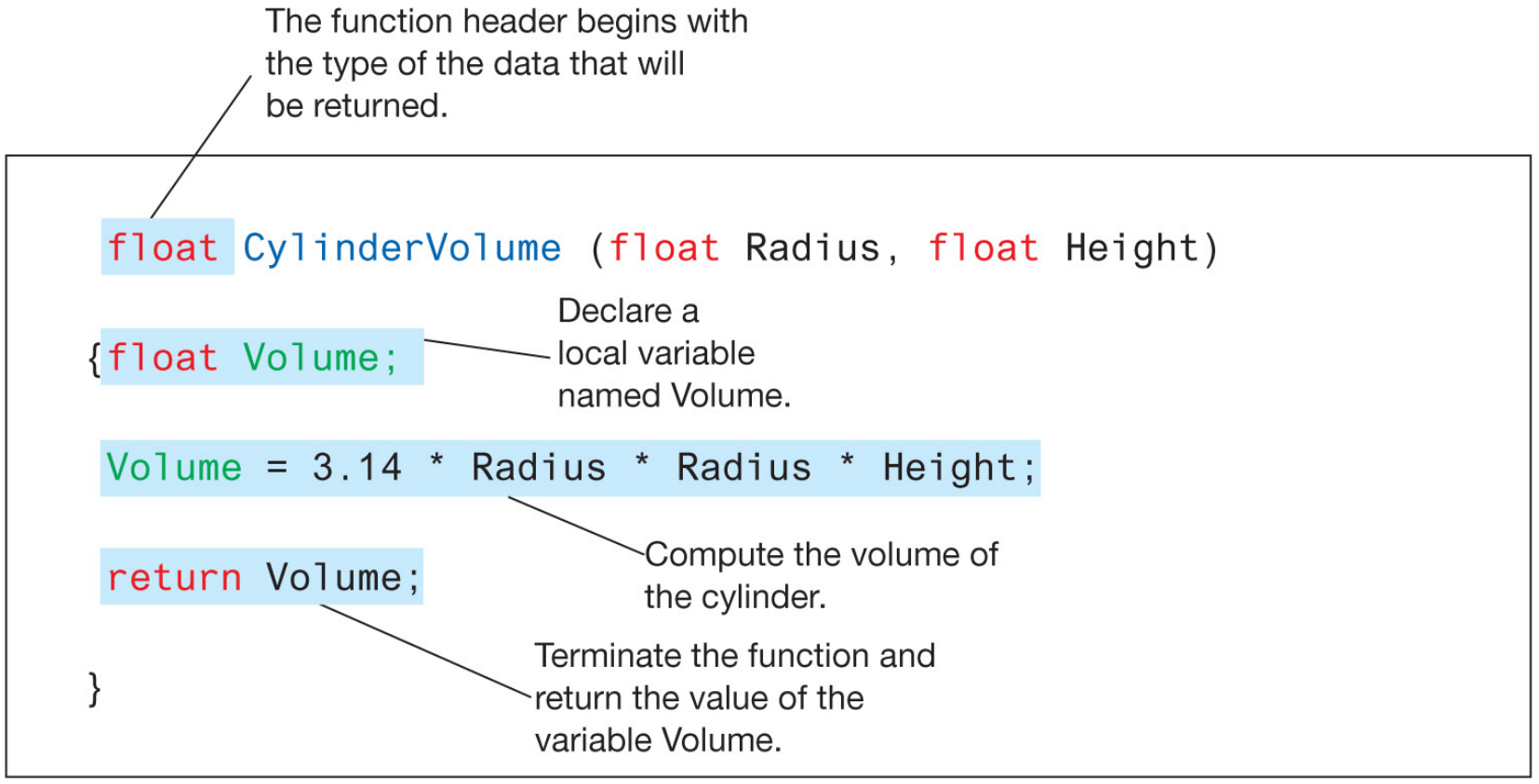


Figure 6.12 The fruitful function CylinderVolume written in the programming language C



6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

```python
# 距離公式
def vt_func(v0, a, t):
    s = v0 * t + (0.5) * a * (t^2)
    return s

# A同學
v0_A = 10
a_A = 20
t_A = 200
s_A = vt_func(v0_A, a_A, t_A)

# B同學
v0_B = 5
a_B = 200
t_B = 100
s_A = vt_func(v0_B, a_B, t_B)

# C同學
v0_C = 50
a_C = 20
t_C = 1
s_A = vt_func(v0_C, a_C, t_C)
```

6.3 Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function
- Unit begins with the function's **header**
- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference

Figure 6.10 Executing the function Demo and passing parameters by value

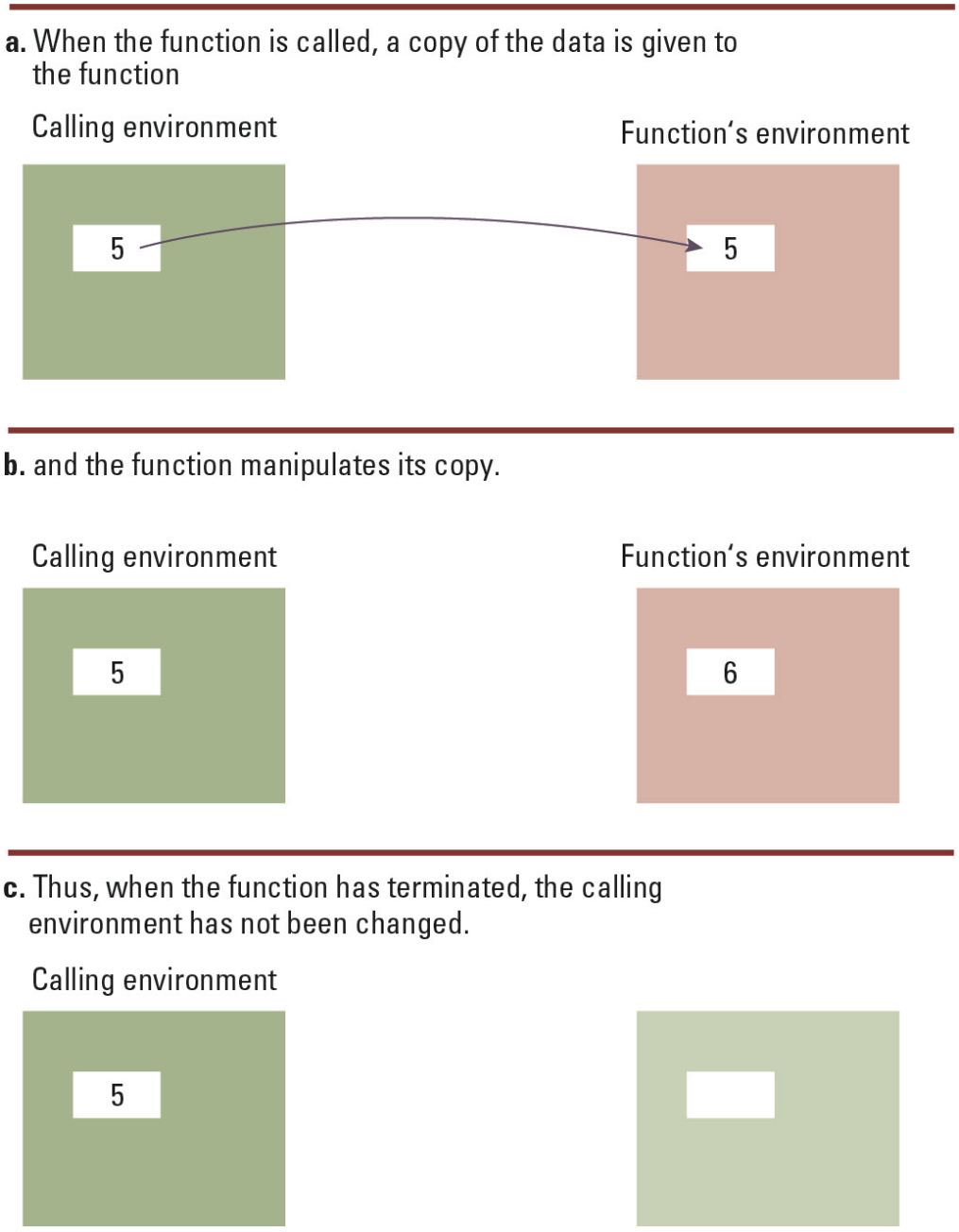
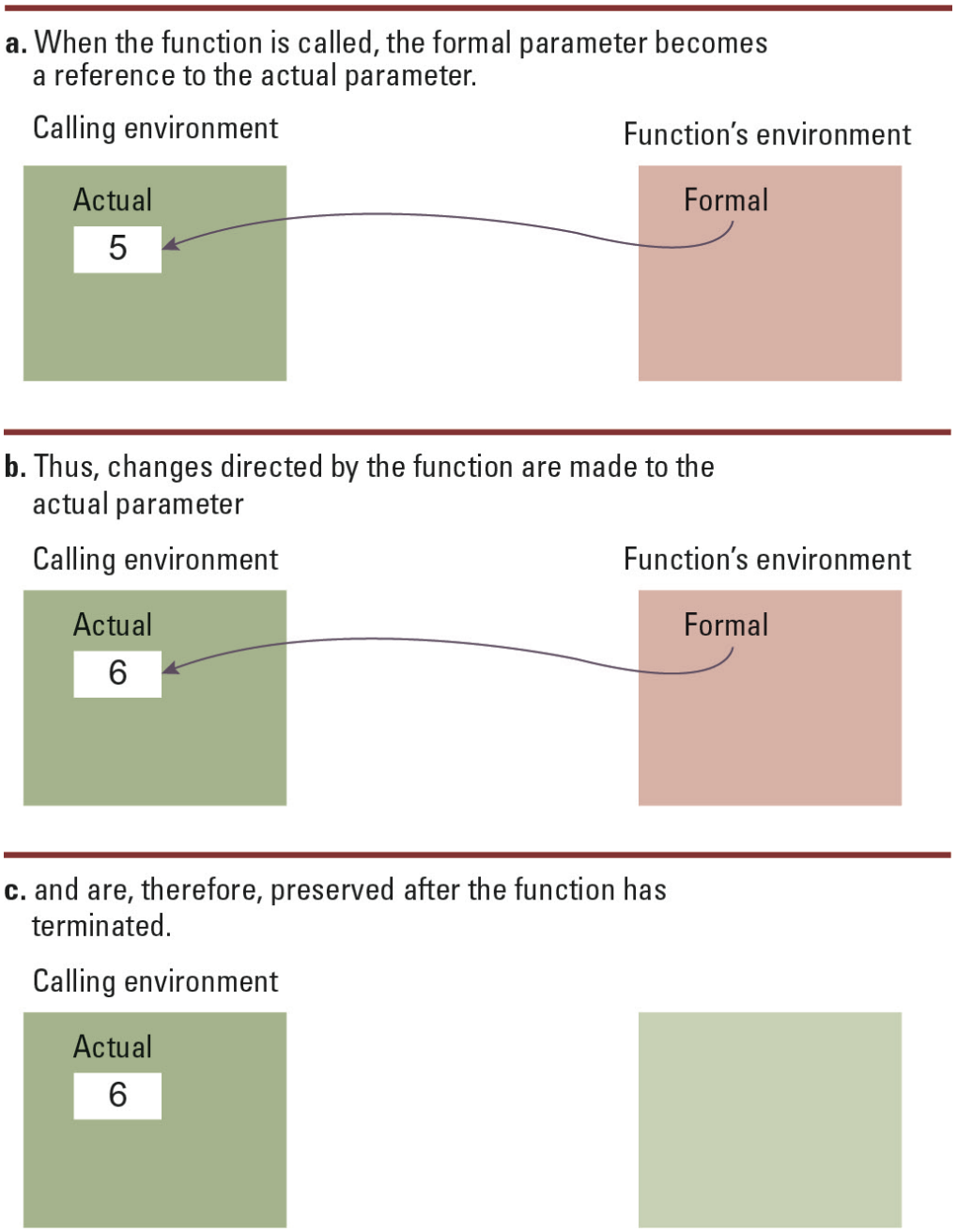


Figure 6.11 Executing the function Demo and passing parameters by reference



6.4 Language Implementation

- The process of converting a program written in a high-level language into a machine-executable form.
  - The Lexical Analyzer recognizes which strings of symbols represent a single entity, or token.
  - The Parser groups tokens into statements, using syntax diagrams to make parse trees.
  - The Code Generator constructs machine-language instructions to implement the statements.

Figure 6.13 The translation process

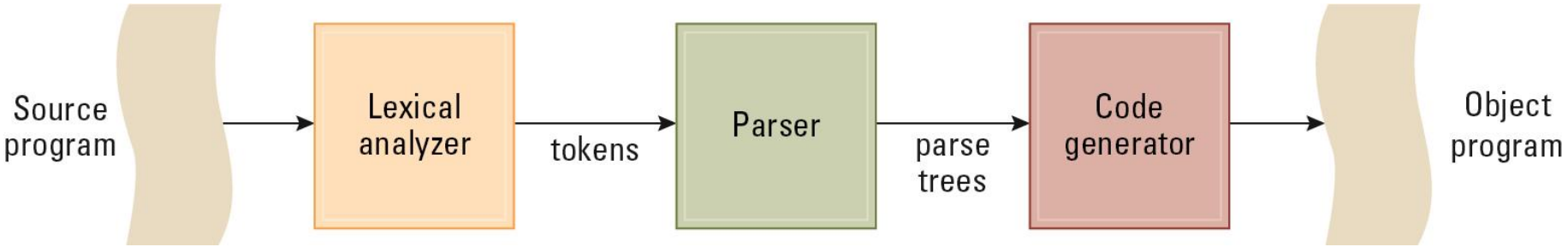


Figure 6.14 A syntax diagram of Python's if-then-else statement

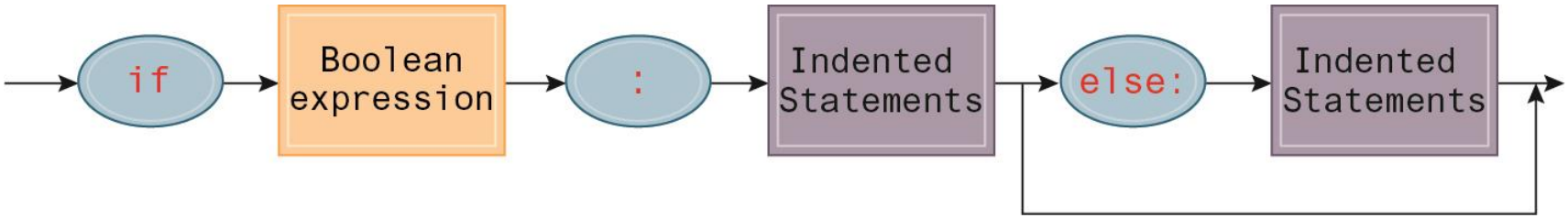


Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression

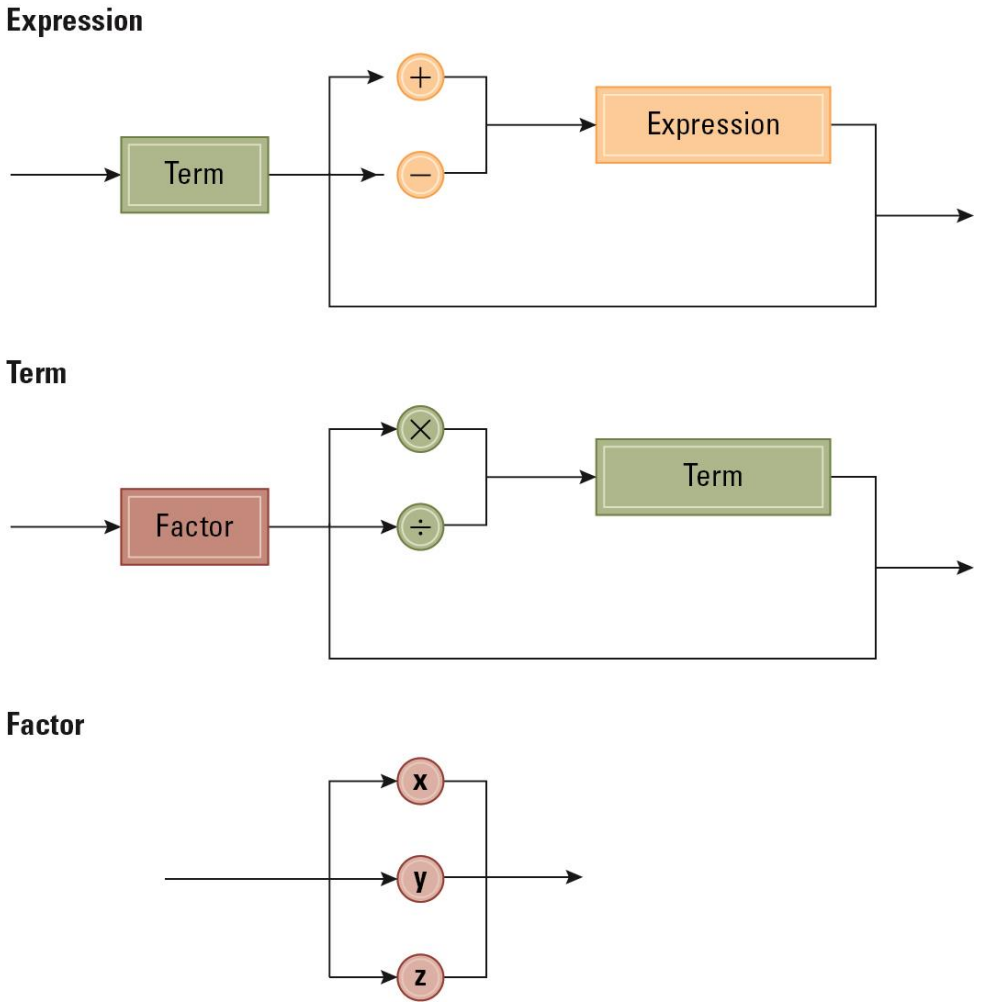


Figure 6.16 The parse tree for the string x + y * z based on the syntax diagrams in Figure 6.17

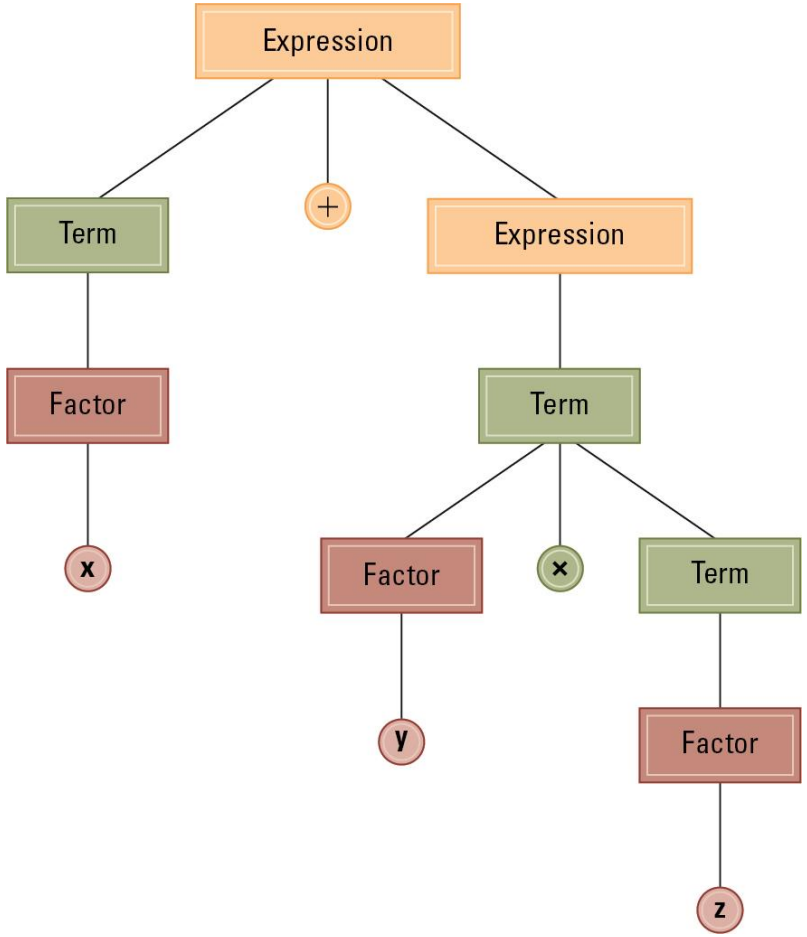
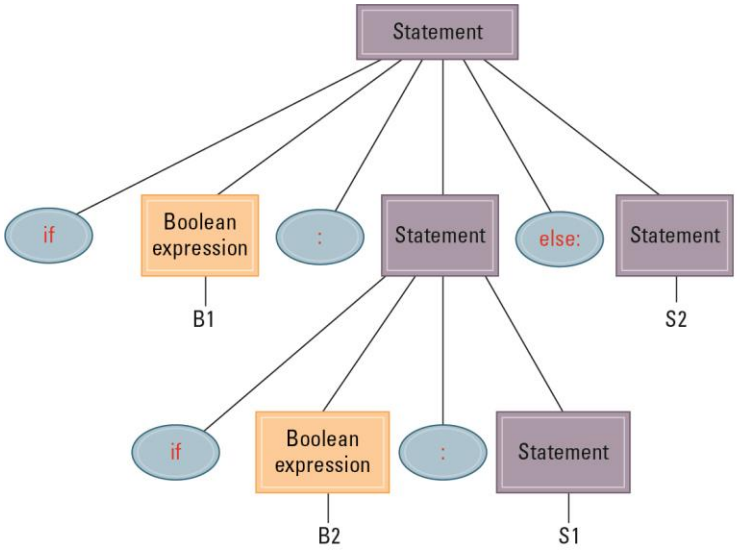


Figure 6.17 Two distinct parse trees for the statement if B1 then if B2 then S1 else S2



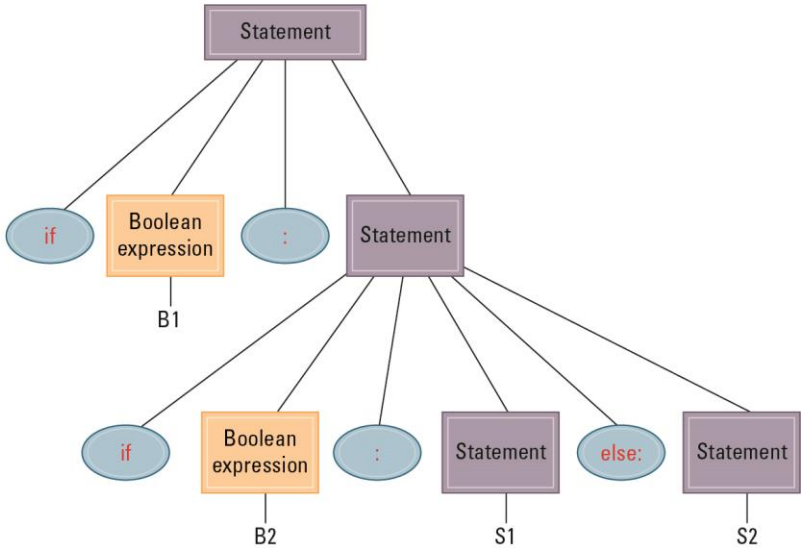
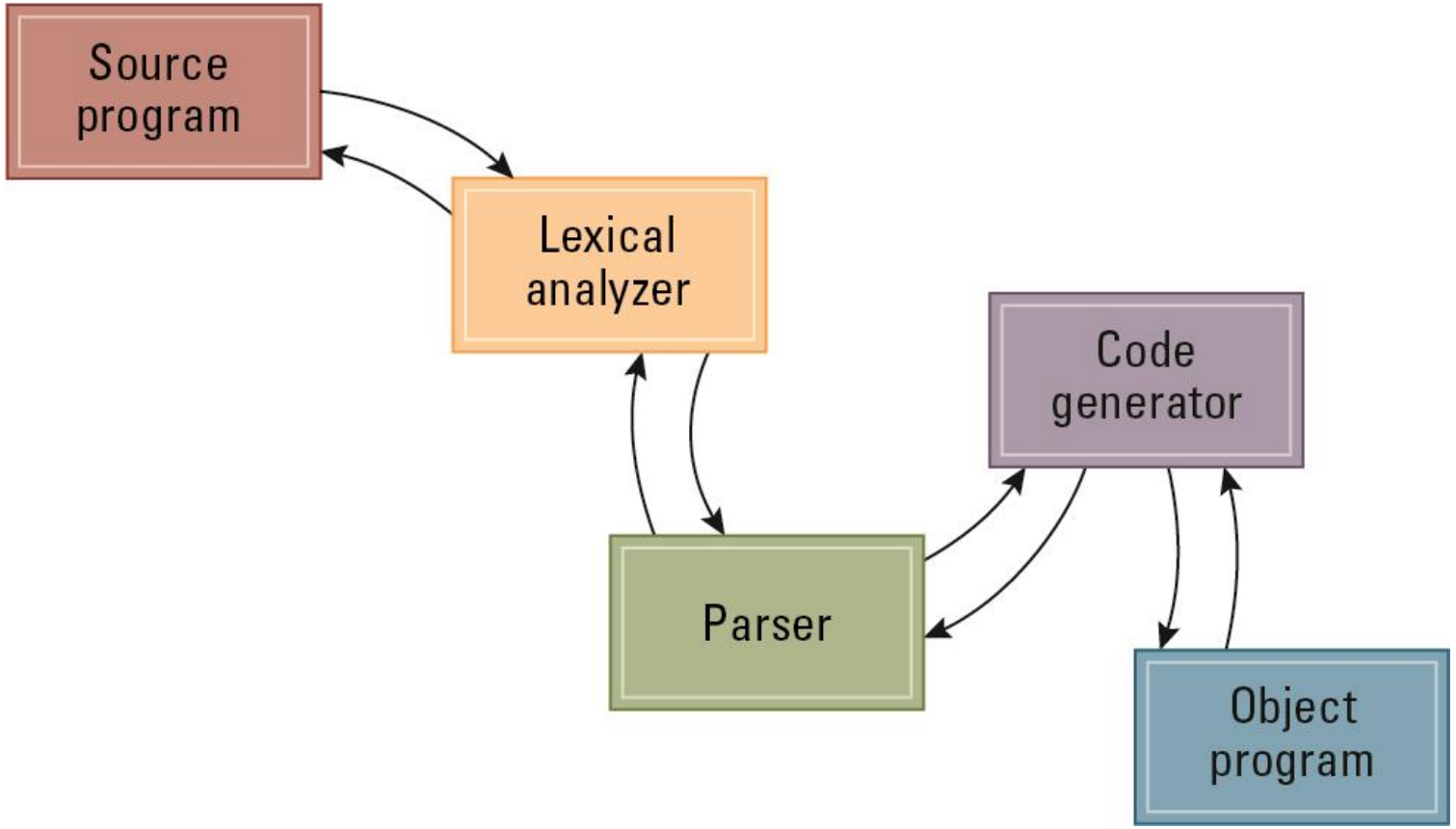


Figure 6.18 An object-oriented approach to the translation process

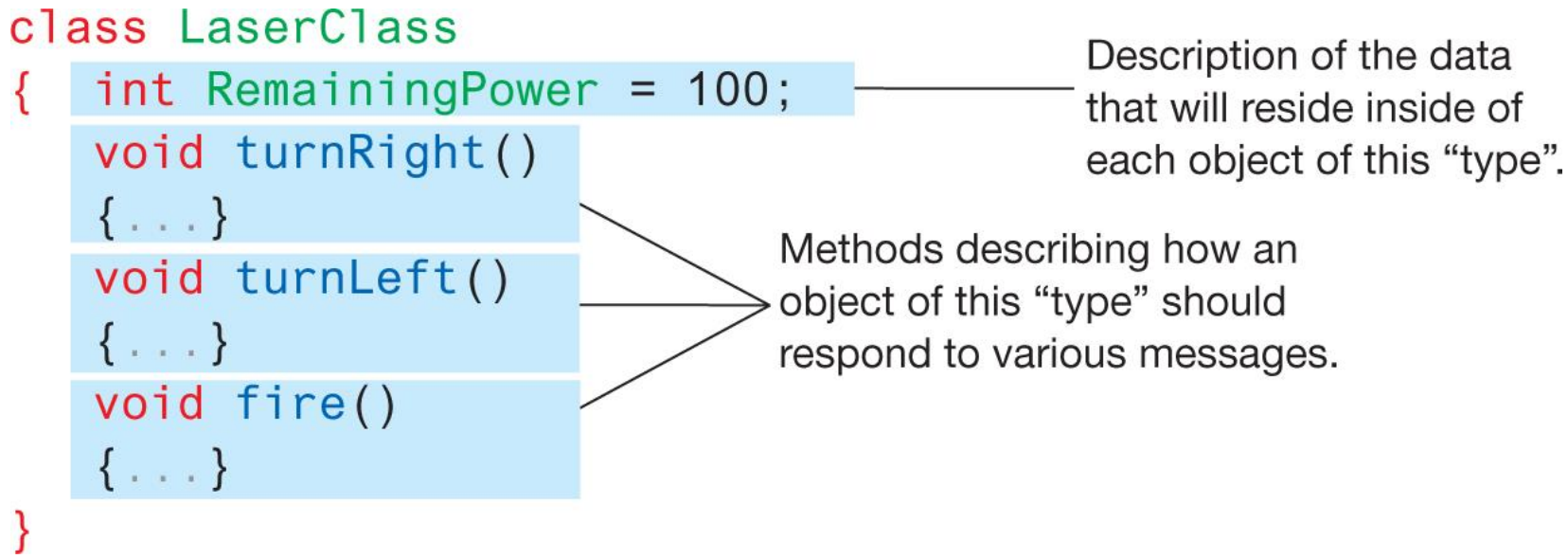


6.5 Object-Oriented Programming

- **Object:** Active program unit containing both data and procedures
- **Class:** A template from which objects are constructed

An object is called an **instance** of the class.

Figure 6.19 The structure of a class describing a laser weapon in a computer game



Components of an Object

- **Instance Variable:** Variable within an object
  - Holds information within the object
- **Method:** Procedure within an object
  - Describes the actions that the object can perform
- **Constructor:** Special method used to initialize a new object when it is first constructed

Figure 6.21 A class with a constructor

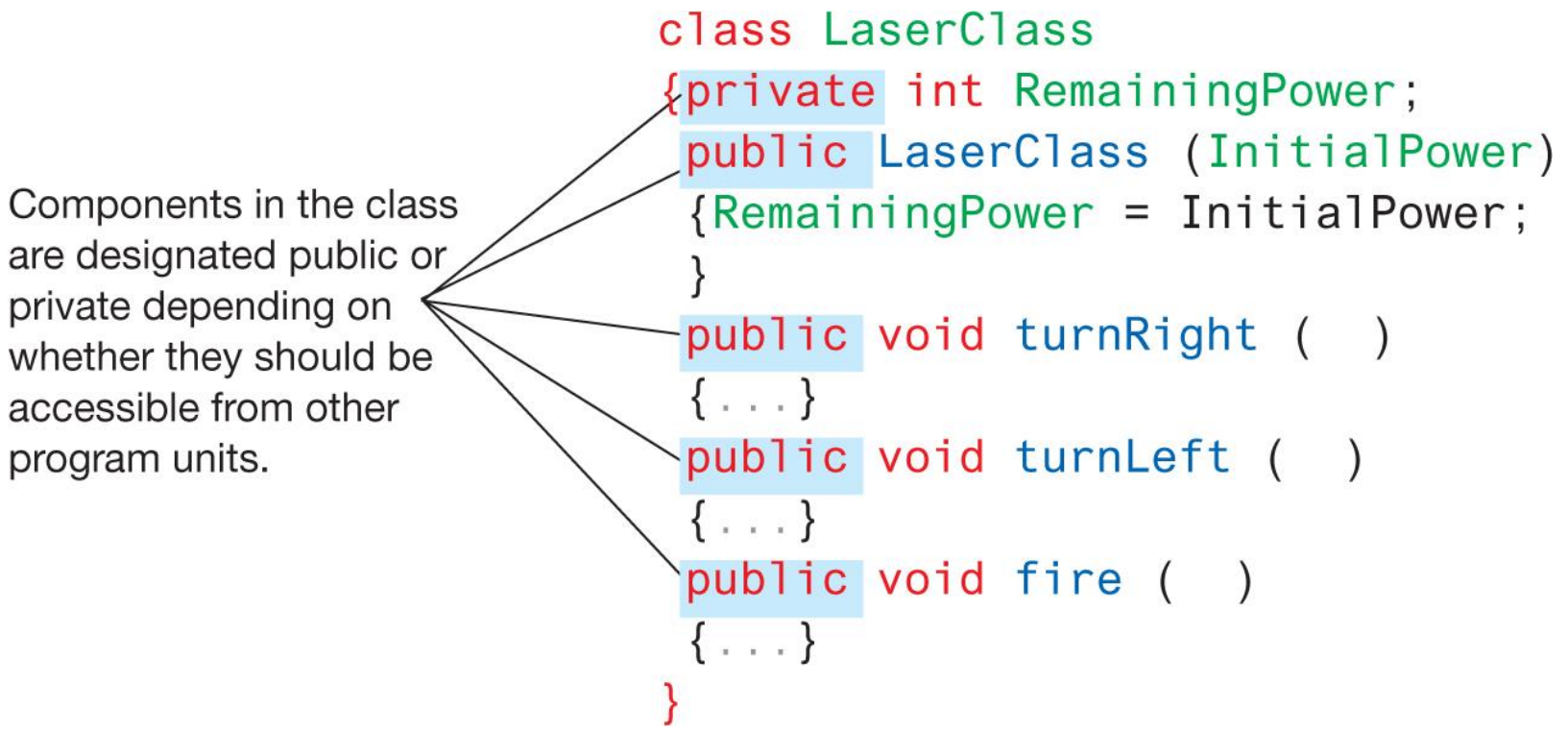
```
class LaserClass
{ int RemainingPower;
  LaserClass(InitialPower)
  { RemainingPower = InitialPower;
  }
  void turnRight()
  {...}
  void turnLeft()
  {...}
  void fire()
  {...}
}
```

Constructor assigns a value to RemainingPower when an object is created.

Object Integrity

- **Encapsulation:** A way of restricting access to the internal components of an object
  - Private
  - Public

Figure 6.22 Our LaserClass definition using encapsulation as it would appear in a Java or C# program



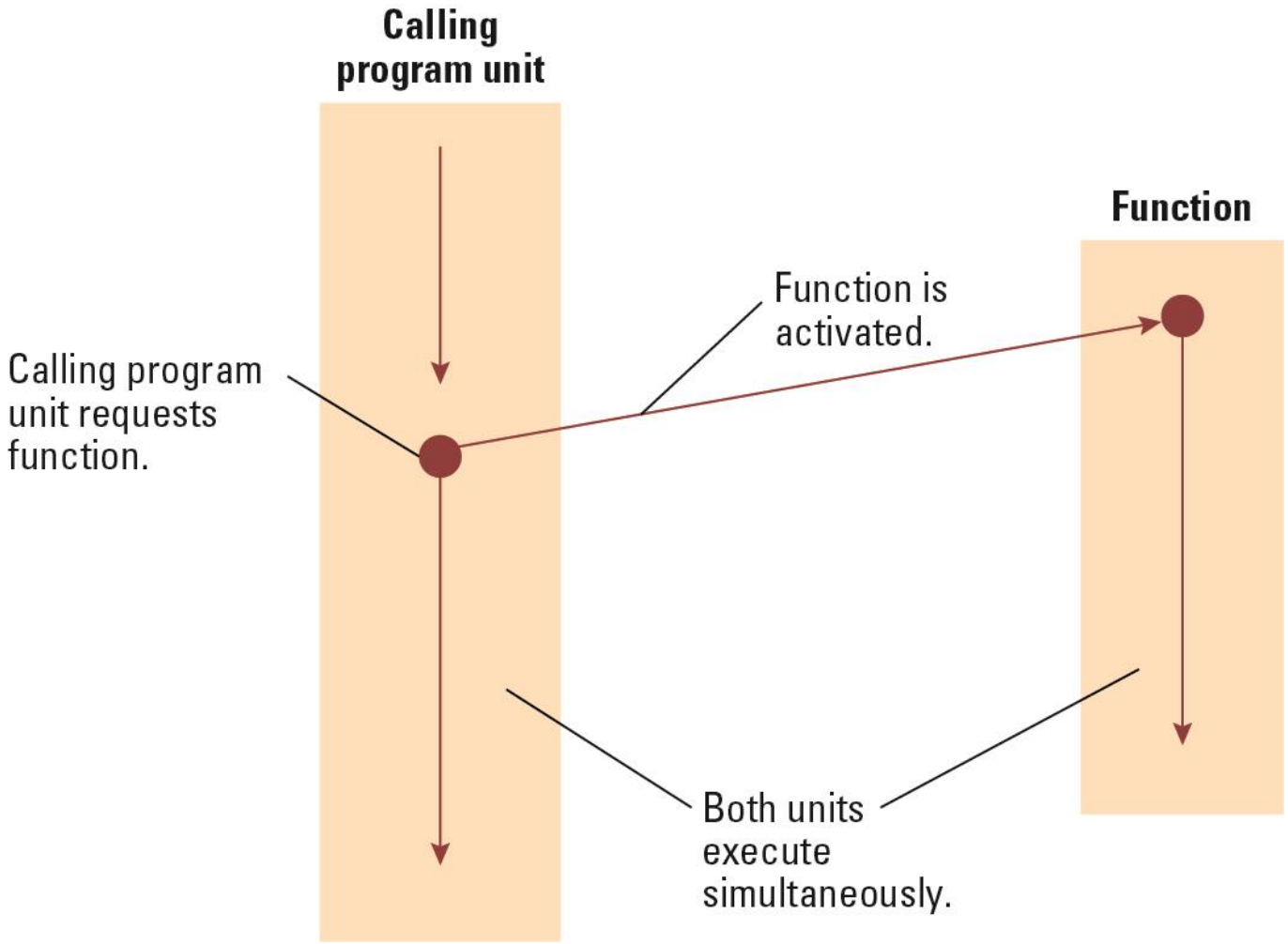
Additional Object-oriented Concepts

- **Inheritance:** Allows new classes to be defined in terms of previously defined classes
- **Polymorphism:** Allows method calls to be interpreted by the object that receives the call

6.6 Programming Concurrent Activities

- **Parallel** (or **concurrent**) **processing:** simultaneous execution of multiple processes
  - True concurrent processing requires multiple CPUs
  - Can be simulated using time-sharing with a single CPU

Figure 6.23 Spawning threads



Controlling Access to Data

- **Mutual Exclusion:** A method for ensuring that data can be accessed by only one process at a time
- **Monitor:** A data item augmented with the ability to control access to itself

6.7 Declarative Programming

- **Resolution:** Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
  - Example: (P OR Q) AND (R OR ¬Q) resolves to (P OR R)
  - **Resolvent:** A new statement deduced by resolution
  - **Clause form:** A statement whose elementary components are connected by the Boolean operation OR
- **Unification:** Assigning a value to a variable so that two statements become "compatible."

Figure 6.24 Resolving the statements (*P* OR *Q*) and (*R* OR ¬*Q*) to produce (*P* OR *R*)

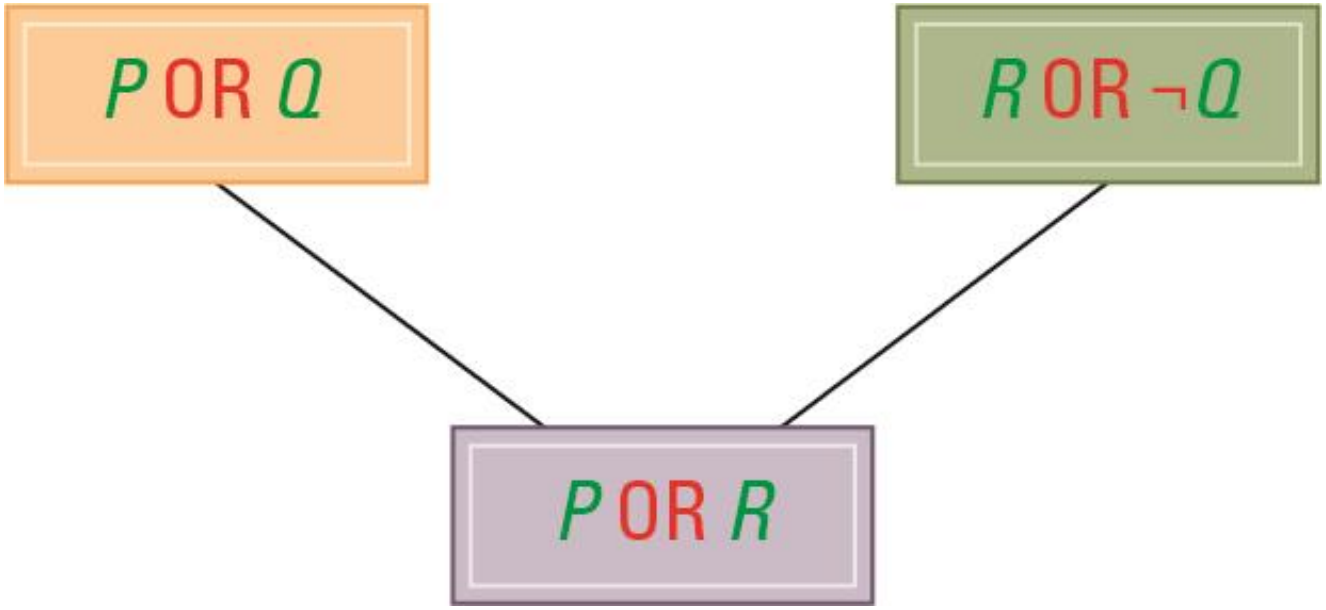
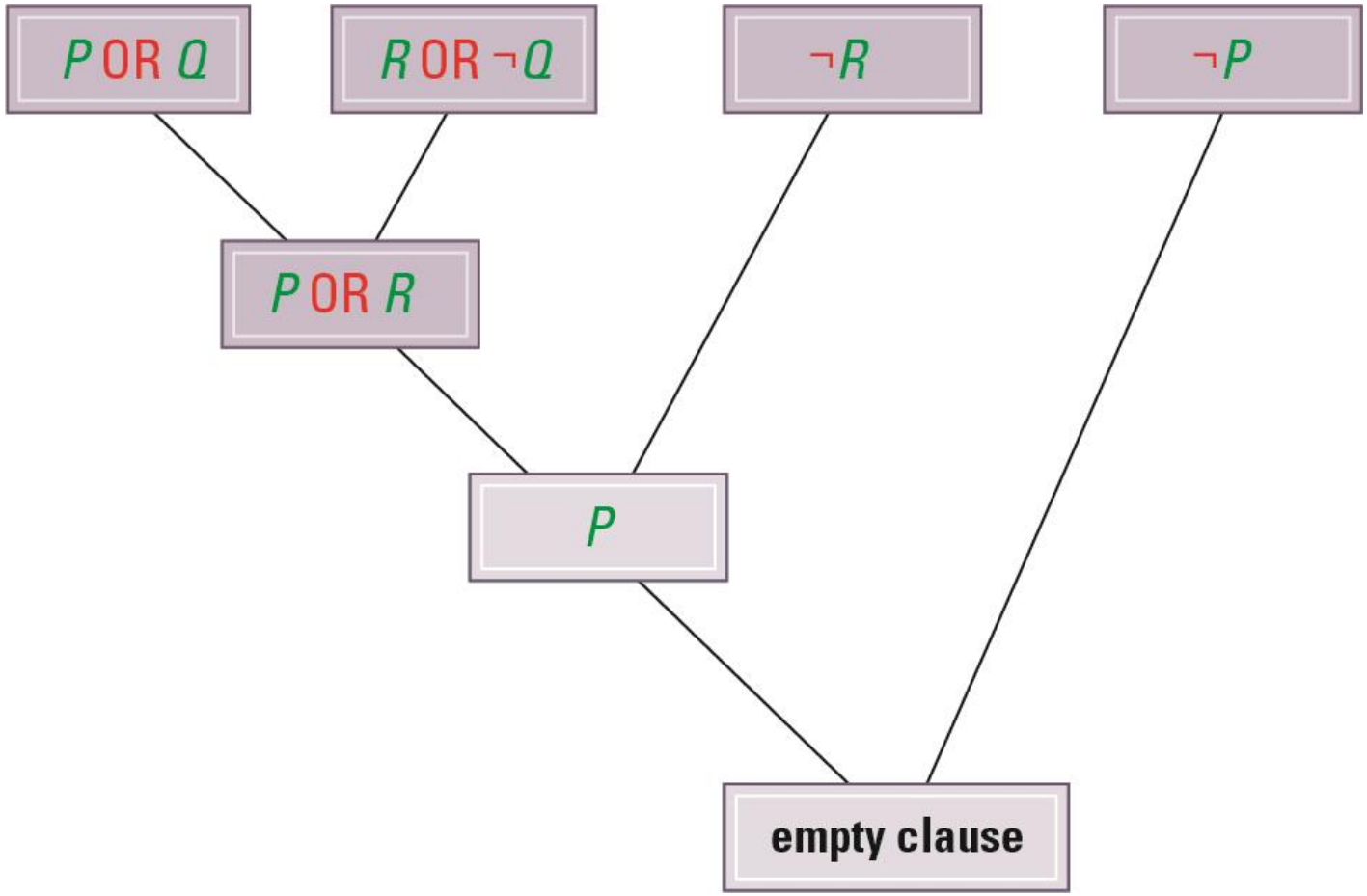


Figure 6.25 Resolving the statements (*P* OR *Q*), (*R* OR ¬*Q*), ¬R, and ¬*P*



Prolog

- **Fact:** A Prolog statement establishing a fact
  - Consists of a single predicate
  - Form: *predicateName*(*arguments*).
    - Example: `parent(bill, mary).`
- **Rule:** A Prolog statement establishing a general rule
  - Form: *conclusion* :- *premise.*
    - :- means “if”
  - Example: `wise(X) :- old(X).`
  - Example: `faster(X,Z) :- faster(X,Y), faster(Y,Z).`